THE UNIVERSITY *of* EDINBURGH

Querying graphs with data

Domagoj Vrgoč

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2014

Abstract

Graph data is becoming more and more pervasive. Indeed, services such as Social Networks or the Semantic Web can no longer rely on the traditional relational model, as its structure is somewhat too rigid for the applications they have in mind. For this reason we have seen a continuous shift towards more non-standard models. First it was the semi-structured data in the 1990s and XML in 2000s, but even such models seem to be too restrictive for new applications that require navigational properties naturally modelled by graphs. Social networks fit into the graph model by their very design: users are nodes and their connections are specified by graph edges. The W3C committee, on the other hand, describes RDF, the model underlying the Semantic Web, by using graphs. The situation is quite similar with crime detection networks and tracking workflow provenance, namely they all have graphs inbuilt into their definition.

With pervasiveness of graph data the important question of querying and maintaining it has emerged as one of the main priorities, both in theoretical and applied sense. Currently there seem to be two approaches to handling such data. On the one hand, to extract the actual data, practitioners use traditional relational languages that completely disregard various navigational patterns connecting the data. What makes this data interesting in modern applications, however, is precisely its ability to compactly represent intricate topological properties that envelop the data. To overcome this issue several languages that allow querying graph topology have been proposed and extensively studied. The problem with these languages is that they concentrate on navigation only, thus disregarding the data that is actually stored in the database.

What we propose in this thesis is the ability to do both. Namely, we will study how query languages can be designed to allow specifying not only how the data is connected, but also how data changes along paths and patterns connecting it. To this end we will develop several query languages and show how adding different data manipulation capabilities and different navigational features affects the complexity of main reasoning tasks. The story here is somewhat similar to the early success of the relational data model, where theoretical considerations led to a better understanding of what makes certain tasks more challenging than others. Here we aim for languages that are both efficient and capable of expressing a wide variety of queries of interest to several groups of practitioners. To do so we will analyse how different requirements affect the language at hand and at the end provide a good base of primitives whose inclusion into a language should be considered, based on the applications one has in mind. Namely, we consider how adding a specific operation, mechanism, or capability to the language affects practical tasks that such an addition plans to tackle. In the end we arrive at several languages, all of them with their pros and cons, giving us a good overview of how specific capabilities of the language affect the design goals, thus providing a sound basis for practitioners to choose from, based on their requirements.

Lay Summary of Thesis

Services such Social Networks (Facebook, Twitter, etc.) and the Semantic Web are becoming more and more ubiquitous in the modern world. What all of these services have in common is that they require new methods for storing and representing their data. Indeed, the classical relational storage model that used to be the staple for storing any type of data seems to be somewhat too rigid to be of use here. Motivated by this we have witnessed a gradual shift towards new models in the previous decades. First it was the semi-structured data and the XML, but both of these approaches are still somewhat limited, so in the previous years the research community started developing the graph data model that naturally lends itself to the aforementioned services.

With new data model also comes the challenge of finding efficient ways of extracting and querying its data. Several languages to query such data have been already proposed, however, all of them come with certain limitations that are connected to the way they view the paradigm of graph data. Namely, they either cling onto the old relational approach and use it to query such data, or they focus on the new features in a way that ignores some important aspects of the relational approach. What we do in this thesis is reconcile the two approaches and propose query languages that take good aspects of both. We develop a sound theory of graph query languages and show how various features affect the language in terms of efficiency and expressive power and at the end provide recommendations that should be useful to database vendors when determining what language is best suited for their specific goals.

Acknowledgements

First and foremost, I would like to thank my supervisor Leonid Libkin for his support and advice during my studies. In addition to allowing me to immerse myself in a colourful and lively scientific environment, he also managed to introduce me to the finest spirits that the Scottish countryside has to offer, on which I am undoubtedly grateful.

Next, I would like to thank Jan Van den Bussche and Wenfei Fan for agreeing to be on my examination committee and for providing many useful suggestions.

I would also like to thank Mladen Vuković, who supervised my studies in Zagreb and introduced me to the area of mathematical logic that finally led me, although following a slightly uneven path, to computer science and database theory.

A special mention goes to Juan for encouraging me in difficult times and suffering through the trouble of writing papers with me. Out of many great people I had the luck to meet during the previous years I am particularly grateful to my other co-authors: Egor, Wim and Tony.

Many thanks also go to Diego, Claire and Myrto for reading parts of this thesis and providing many helpful comments.

Finally, I would like to thank friends and family for their support.

This work and my studies were made possible by the generous support of EPSRC grants G049165 and J015377, as well as FET-Open Project FoX, grant agreement 233599.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Domagoj Vrgoč*)

Table of Contents

1 Introduction **1**

1.1 Graph databases and their languages 1

1.2 Contributions 4

1.3 Other related work 9

2 Preliminaries **11**

2.1 Graph databases 12

2.2 Regular path queries and extensions 15

2.3 Nested regular expressions 17

2.4 Query evaluation 19

2.5 Path languages and Graph languages 20

I Path languages **25**

3 From words to paths **27**

3.1 Data words vs data paths 29

3.2 Ruling out bad alternatives 31

4 Languages for data paths **35**

4.1 Register automata as a query language 37

4.2 Regular queries with memory (RQMs) 42

4.3 Regular queries with binding (RQBs) 53

4.4 Regular queries with data tests (RQDs) 59

4.5 Variable automata 67

4.6 Summary of complexity results 74

5 Additional features **75**

5.1 Languages with inverse 76

5.2 Conjunctive queries 81

5.3 Adding variables to register automata 83

6 The language theory gap **89**

6.1 Register automata . . . 92
6.2 Regular expressions with memory . . . 98
6.3 Regular expressions with binding . . . 107
6.4 Regular expressions with equality . . . 119
6.5 Variable automata . . . 126
6.6 Summary of language theoretic properties . . . 130

II Graph languages and beyond **133**

7 Graph XPath **135**

7.1 The language and its many variants . . . 137
7.2 Query evaluation . . . 143
7.3 Expressive power . . . 146
7.3.1 Expressiveness of navigational languages . . . 147
7.3.2 Expressiveness of data languages . . . 156
7.4 Hierarchy of the fragments . . . 160
7.5 Conjunctive Graph XPath queries . . . 163
7.6 Summary . . . 164

8 Beyond graphs – TriAL **167**

8.1 Graph databases and RDF . . . 169
8.2 An Algebra for RDF . . . 175
8.3 A Declarative Language . . . 180
8.4 Query Evaluation . . . 184
8.5 Low-complexity fragments . . . 188
8.6 Expressive power . . . 195
8.7 Summary . . . 207

III Analysing the languages: Comparison and Containment **209**

9 Comparing the languages **211**

9.1 Path queries . . . 211
9.2 Moving up the food chain . . . 212
9.3 Triple algebra and graph languages . . . 217
9.4 The complete picture . . . 224

10 Query containment **227**
10.1 Containment of path queries 229
10.1.1 Containment of RQMs 230
10.1.2 Containment of RQDs 237
10.1.3 Impact of inverse on containment 240
10.1.4 Containment of Variable automata 243
10.2 GXPath and its many fragments 244
10.2.1 Containment of navigational languages 244
10.2.2 Containment with data values 252
10.2.3 Coming back to the core 253
10.3 Summary of containment results 254

IV Wrapping up **257**

11 Conclusions and future work **259**
11.1 Choosing the right language 260
11.2 Where to go from here 263

Bibliography **267**

Index **276**

Nemiri
Zaboravi nemire
Blaise Cendrars

Chapter 1

Introduction

1.1 Graph databases and their languages

In recent years we have witnessed a renewal of interest in managing and maintaining graph structured data, motivated by a high demand from services that find the traditional relational model too restrictive. The the origins of the graph data model can be traced back to the 1960s and the network model used by Charles Bachman as a template for designing one of the first general-purpose database management systems called Integrated Data Store and developed at General Electric [Bachman, 1973]. With the emergence of relational databases the model was then abandoned in the seventies and early eighties, but was again revisited during late eighties [Cruz et al., 1987, Consens and Mendelzon, 1990], when it was used for searching and storing hypertext systems [Consens and Mendelzon, 1989], and started regaining popularity with the prominence of semi-structured data in the 1990s [Abiteboul et al., 1999]. However, its full potential only became apparent with the emergence of the Semantic Web [W3C Consortium, 2013, Pérez et al., 2010, Gutierrez et al., 2011] and Social networks [Ronen and Shmueli, 2009, San Martín and Gutierrez, 2009, Fan, 2012], where the data is naturally represented in a graph like structure [Klyne and Carroll, 2004]. Other applications of the graph data model also include crime detection networks [Fan et al., 2010b, Fan et al., 2010a], biological databases [Olken, 2003, Leser, 2005, Milo et al., 2002] and querying workflow and data provenance [Anand et al., 2010, Dey et al., 2013]. As a result of this there are now several vendors offering graph database products [Neo4j, 2013, Dex, 2013] and a steady stream of research literature on the subject (for a survey see e.g. [Angles and Gutierrez, 2008, Barceló, 2013, Wood, 2012]).

In all of these applications data is modelled by a graph, with nodes representing entities in the database and edges representing various connections these entities can form. For example if we are describing a social network it is natural to represent users by nodes, with edges symbolizing the connection between two users, such as friends, co-workers, relatives and so

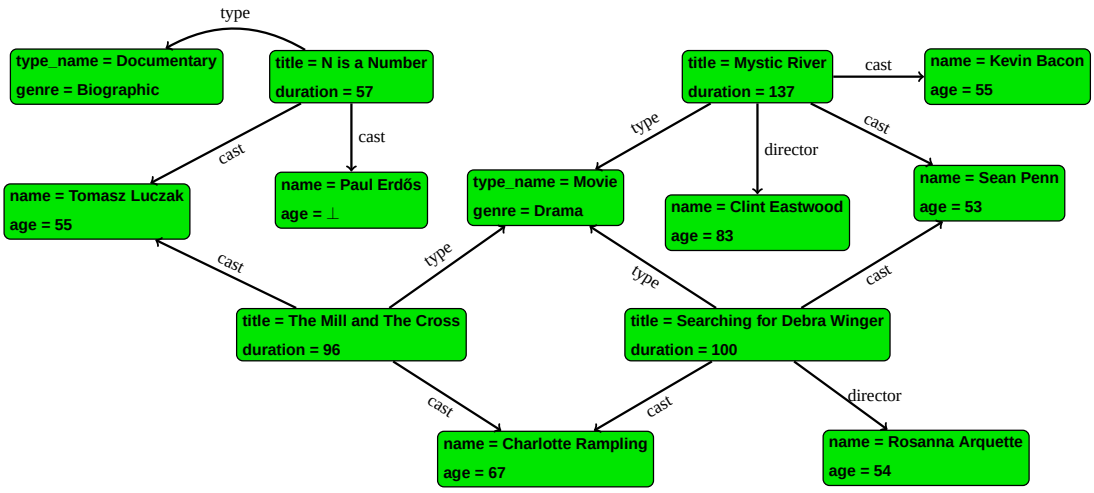


Figure 1.1: A movie database represented as a graph

on. Another example would be a movie database where each node stores information about a specific movie, movie genre, or actor, while the edges of the graph tell us how two entities are connected. We could for instance have an edge between a node representing a specific actor and a node representing a movie the actor had starred in, or an edge connecting a movie with its generic description. One such database is presented in Figure 1.1. Since nodes can form different types of connections, it is usual to assign labels to the edges connecting them. Finally, nodes themselves contain the actual data, such as the information about the movie title and duration, actor's names and ages, etc. The data is of course modelled as the usual relational data with attribute values coming from an infinite domain [Angles and Gutierrez, 2008].

One of the fundamental issues related to graph data is of course the question of querying it. When designing query languages one is primarily concerned with striking a good balance between expressivity and efficiency. Namely, a language has to be capable of describing a wide variety of relevant queries , while at the same time keeping the complexity of main reasoning tasks low. To achieve this for graph data two separate approaches have been studied in the past.

The first approach treats the graph model as a relational database and uses traditional relational languages to extract the data. For example in the database above one could ask for all movies of the same duration, or all actors of the same age. The class of queries one typically uses to express such properties is the class of *conjunctive queries* [Abiteboul et al., 1995].

On the other hand, what makes graph databases attractive in modern applications is the ability to query intricate navigational patterns between objects, thus obtaining more information about the *topology* of the stored data. For example, considering the database in Figure 1.1 one might want to find pairs of actors connected by collaboration connections. This query would give us that Paul Erdős and Charlotte Rampling have collaborated since they both co-starred with Tomasz Luczak. The same can be said for Kevin Bacon and Paul Erdős, but the sequence

of collaborations is now longer. Taking into consideration that our databases can grow by inserting more data, it is easy to see that no fixed number of collaborators can be set in advance to answer this query, thus calling for languages that allow full transitive closure. A basic building block for such languages are typically *regular path queries*, or *RPQs*, that select nodes connected by a path described by a regular language over the labelling alphabet [Cruz et al., 1987]. Extensions of RPQs with more complex patterns, backward navigation and relations over paths have been studied extensively too [Abiteboul and Vianu, 1999, Barceló et al., 2012a, Barceló et al., 2012b, Calvanese et al., 2000, Calvanese et al., 2009, Consens and Mendelzon, 1990].

Note that both of these approaches treat the data and the topological patterns enveloping it as two separate entities. Thus, the querying mechanisms one deals with generally fall into one of the following categories:

- queries about *data*, i.e., essentially relational queries (e.g., finding pairs of actors of the same age), or
- queries about *topology* such as finding nodes connected by a path with a certain label (e.g., actors who are connected via collaboration links).

However, both approaches have some serious shortcomings. As mentioned above, treating the graph model as a relational database, while allowing to extract information about the stored data, completely ignores topological queries that explore various patterns connecting the data. On the other hand, traditional graph languages such as RPQs and their extensions talk only about the topology, while ignoring the data. What both of these approaches are incapable of doing is *combining data and topology*. As an example of a query that involves such a combination, one could for example ask for people who have a finite Bacon number (that is, there is a sequence of collaboration connections linking them with Kevin Bacon). Note that here we have to test that the name attribute of the final actor in the sequence is indeed Kevin Bacon and not some arbitrary value. Another example is a query that finds actors connected via professional links restricted to actors of the same age. In this case, comparison of data values (having the same age) is done for every node along the path. A similar query might ask for people with a finite Bacon number, but such that collaboration connections must always go through movies – documentaries will no longer suffice. In our example this would still give us that Tomasz Luczak has a finite Bacon number, but Paul Erdős does not, because his connection is realized by co-starring in a documentary.

Since answering such queries lies at the very core of many applications using the graph data model, this opens up space for the main focus of this dissertation which is the design and analysis of languages for querying graph data in a way that allows combining navigational patterns with the data they connect. To this end we will propose several languages, based on traditional and new approaches and explore how they stack one against the other, as well as how

they relate to previously proposed languages, both relational and graph-oriented. The purpose of such a study is, of course, to point to a good set of primitives that should be present in any graph query language, either theoretical or applied. In the end we will describe several such sets and argue why they could serve as a logical core of a good language for querying graph data.

1.2 Contributions

Describing *the* graph language, both efficient and expressive enough to capture a combination of data and navigational queries, is a difficult task, especially taking into consideration that different groups of users might have different requirements when it comes to the type of queries they wish to ask. The main contribution of this dissertation then is to develop several classes of query languages for graphs with data and to analyse how adding various data manipulation capabilities and navigational features affects the efficiency of the main reasoning tasks such as query evaluation and query containment, as well as how it relates to the expressive power of the language.

To explain two main design principles for the languages we propose it is important to notice the duality present in traditional graph languages that disregard data values and only reason about edge labels. To illustrate this consider for instance RPQs, a standard building block for any navigational language over graphs. An RPQ query is specified by a regular expression and it retrieves all pairs of nodes connected by a path whose edge labels form a word belonging to the language defined by this expression. Therefore, in this context one uses a language theoretic formalism to specify the set of allowed path labels and then searches for a path in the graph whose label belongs to this set. We call such languages *path languages*. On the other hand more advanced languages, such as *nested regular expressions*, or NREs ([Pérez et al., 2010]), work directly on graphs, allowing to search for patterns that can no longer be described by paths alone. Such a query could for instance check if in a sequence of collaborating actors from the example above each movie appearing on the path connecting them has a director entered into the database. These languages will be called *graph languages*. Following this duality we will be talking about path languages and graph languages when considering data values as well.

Path languages We start with the more traditional path based approach and consider various language theoretic formalisms that allow for data values in addition to a finite set of labels. The question then is how to select the one appropriate for the task of querying data graphs? Here we will be governed by the usual objective of keeping the complexity of the query evaluation problem – that is the problem of determining if an object belongs to the answer of a particular query – low. This will allow us to immediately rule out several well studied formalisms, such

as FO [Bojanczyk et al., 2011], Pebble automata [Kaminski and Tan, 2008], or LTL with freeze quantifiers [Demri and Lazić, 2009], leaving us with the model of *register automata* [Kaminski and Francez, 1994], which we modify for our purposes. The class of queries defined by register automata, called *regular data path queries*, or RDPQs for short, has reasonable complexity bounds, in fact matching those of the usual relational languages, and relatively high expressive power, at least when specifying properties of paths is concerned. Its main shortcoming, however, is the relatively cumbersome and unintuitive syntax that is unlikely to attract much interest from the practitioners.

In order to overcome this, we develop an expression analogue of register automata called *regular expressions with memory*. These expressions have the same relationship with register automata as ordinary regular expressions do with NFAs, that is they define the same class of languages, but are much easier to read and specify. To mimic registers they will use variables, allowing one to store a value into a variable in the same way as it would have been stored in a register. The class of queries they give rise to, called *regular queries with memory* (*RQM*s) retain the PSPACE complexity bound of the RDPQ query evaluation problem (dropping to NLOGSPACE if the query is fixed – also known as data complexity in the literature [Vardi, 1982]). This, coupled with easy and intuitive syntax makes them much more useful than register automata as a graph query language.

To lower the complexity of the query evaluation problem we then look into various ways of restricting register automata or *RQM*s, while still retaining most of the expressive power that powerful data manipulation mechanisms used there allow. Examining regular expressions with memory one immediately notices that they do not define proper scope of variables – a feature very common in programming languages and software verification. It is therefore natural to look at a restriction that limits this. By giving variables scope we arrive at the class of *regular queries with binding*, or *RQB*s. Surprisingly, it turns out that the complexity of query evaluation remains the same, although the language has slightly weaker expressive power.

So far we only considered languages operating with variables or registers explicitly, granting high expressive power in terms of data value comparisons. In order to develop an efficient, yet expressive language, we turn to a class of queries that allows testing for data value (in)equality at the beginning and the end of a subpath only. This first-in-last-out discipline will allow us to obtain very low combined complexity (PTIME to be more precise), while still being able to express many interesting graph queries. The class of queries is called *regular queries with data tests* and will be important in understanding how data tests in query languages relate to the ones in first-order logic.

Finally, in order to develop a language that still has the ability to store data values in variables, but at the same time has query evaluation complexity bellow the one of *RQM*s we turn to *variable automata*. We extend this formalism, introduced first in [Grumberg et al., 2010a]

to reason about words over infinite alphabets, to work over data graphs. The complexity here is reasonable, namely *NP*-complete and the different nature of data comparisons that such automata use makes them orthogonal to the previously proposed languages.

An important issue in query language design is enriching the base theoretical languages with features required from database practitioners. In the context of graph databases two of the most important such features are the ability to traverse edges backwards and allowing conjunctive queries to be formed from simple graph queries. Indeed, it has been argued before [Calvanese et al., 2000, Calvanese et al., 2003] that the inverse operator is a required feature of any practical graph language, while the usefulness of conjunctive queries has been well studied both on relational databases [Abiteboul et al., 1995] and on graphs [Barceló et al., 2012b, Freydenberg and Schweikardt, 2011, Bienvenu et al., 2013]. We therefore study the impact of such extensions on previously proposed languages. It turns our that adding inverses has no impact on the complexity of query evaluation (however, it will turn our to have a big impact on query containment, as we later show), and results on conjunctive queries are the best possible in light of the results for more restricted languages.

Overall, we see that as far as path languages are concerned, powerful data manipulation features come with a price of relatively high complexity. This is also coupled with relatively poor navigational power, since such languages can only define paths. We address this issue when defining graph languages.

The results on path languages are presented in Chapters 3 and 4. Extending the languages with inverses, conjunction and the ability to use variables in a more general way is presented in Chapter 5. Most of these results appeared previously in [Libkin and Vrgoč, 2012b] and [Libkin et al., 2013c]. Languages with inverse were briefly studied in [Kostylev et al., 2014] and conjunctive queries for some classes were considered in [Libkin and Vrgoč, 2012b]. Note that in this dissertation all of the languages come equipped with the ability to check equality with a constant, which was not present in the aforementioned sources.

Language theoretic aspects of path languages As mentioned previously, to define path languages, one uses a language theoretic formalism to specify the set of allowed path labels. To properly analyse such languages one must also understand basic properties of formalisms defining them. Indeed, we will later use language containment problem to infer results on query containment of path languages, and the results on how the languages compare one to another over graphs will follow from the study on the expressivity of their language theoretic counterparts.

Since we introduced several new language theoretic formalism in Chapter 4, it is important to understand their properties. We do this in Chapter 6, where we consider these formalisms in the setting of data words (basically words that in each position carry a letter from a finite

alphabet and a data value from an infinite domain) and determine the complexity of basic algorithmic tasks such as membership, nonemptiness or containment for them. We also look at usual closure properties and show how formalisms compare one to another in terms of expressive power.

Most of the results from Chapter 6 have previously appeared in [Libkin and Vrgoč, 2012a, Libkin et al., 2013c]. Some of the result that were missing in these publications are presented here for the first time.

Graph languages Having considered several languages using the traditional path-based approach, in Chapter 7 we turn our attention to languages operating directly over graphs. Extending the idea of nested regular expressions [Pérez et al., 2010], as well as some previous work based on algebras for binary relations [Fletcher et al., 2012, Fletcher et al., 2011], we show how a well established query language from the XML context, namely XPath, can be adapted to suit our purposes. Using the branching capabilities of such languages, coupled with data value tests, we can now for instance search for all the actors in our sample data graph from Figure 1.1 who have a finite Bacon number, but stipulate that the connection is made by co-starring in movies and not documentaries. Tomasz Luczak is then still an answer to our query, however, Paul Erdős is not, since his link to Kevin Bacon goes through the documentary "N is a Number".

The language we propose is called GXPath and we study its query evaluation properties and connections to logic. We obtain good complexity bounds (namely PTIME for any reasonable variant of the language), as well as the ability to express many queries of interest in the graph setting. We also show that the language is strongly rooted in logic, as it is equivalent to an extension of FO with binary transitive closure over graphs. This, together with the fact that its navigational fragment is just PDL [Harel et al., 2000] in disguise, makes GXPath a definitive graph language when navigation is the main priority, and a strong candidate for practitioners to consider when choosing the appropriate language for their purposes. Its main deficiency, the inability to freely use memory in a way that, for instance, *RQM*s do, is somewhat lessened by the fact that XPath-style data tests that the language uses have been tried and tested over time by XML practitioners, however, there are still some properties that *RQM*s can express that are outside the scope of GXPath.

Note that most of the results from Chapter 7 appeared previously in [Libkin et al., 2013a]. Some results, such as the complete hierarchy of the language fragments, connections to FO with data value tests, and conjunctive queries based on GXPath, are however new and appear here for the first time.

RDF and graph data RDF databases are often cited as one of the most important application of the graph data model, however, there is a slight mismatch between data graphs and RDF

Triplestores. Although big majority of RDF data is indeed a graph, the model itself allows edge labels to be objects themselves, thus permitting them to be a source of another edge. This fact becomes increasingly important in areas such as data integration, provenance tracking, or querying and maintaining clustered data.

In Chapter 8 we develop a language with such applications in mind. On top of that, we design the language specifically for RDF data, thus making it closed in the same way that relational algebra never takes the user outside of the relational model. The language, called TriAL, is based on the concepts of relational algebra, but also allows a limited amount of recursion. Here we study the usual query evaluation problem, compare the language to previously proposed languages for RDF (namely SPARQL and nSPARQL [Harris and Seaborne, 2013, Pérez et al., 2010]) and compare it to traditional relational languages. We also show that, due to its close connections with relational algebra, the language has a well defined Datalog equivalent, making it very attractive to the users. The main conclusion here is that treating RDF data model as a graph database has some inherent limitations and considering it in full generality leads to a richer theory, subsuming that of graphs alone. On the other hand, this study also allows to transfer RDF techniques back to graphs, allowing more general navigational and data patterns.

Most of the results appearing in Chapter 8 were previously presented in [Libkin et al., 2013b].

Comparing the languages To obtain a complete picture of the graph querying landscape, in Chapter 9 we compare previously introduced languages in terms of expressive power. As it turns out, the ability to use variables makes path languages incomparable to the navigationally richer query classes such as GXPath or TriAL. On the other hand, variable automata turn out to be orthogonal to all of the other languages because of their somewhat unnatural ability to guess assignments beforehand, thus giving them the ability to reason globally, unlike the other languages which are based on the automata or expressions that are in essence local.

Although most of the results in Chapter 9 have already appeared in the publications where the languages were originally introduced (see above), some of the results are new and have not been considered previously.

Query containment Finally, in Chapter 10 we will initiate the study of static analysis aspects of our languages. Here we concentrate on the problem of query containment which asks us to determine, given two queries in some language, if the answer set of the first query is contained in the answer set of the second query over all possible data graphs. Query containment is a fundamental database theory problem [Abiteboul et al., 1995] and is crucial in several important database tasks such as query optimisation, view definition and maintenance and view-based query answering. In this chapter we study the problem for previously proposed languages and

determine the decidability border based on both data manipulation abilities as well as navigational features a language allows. It turns out that decidability can not be established without severe restrictions on the use of negation and data inequalities, but once these are excluded from the language we generally obtain reasonable complexity bounds, ranging from PSPACE to EXPSPACE. While we obtain a relatively complete picture for the class of path languages and several of their extensions, the situation is far from being resolved in the case of GXPath, where the abundance of fragments promises to be a fruitful ground for future research, similarly as was the case with XPath over trees [Figueira, 2010b].

Most of the results presented in Chapter 10 already appeared in [Kostylev et al., 2014], however some results, such as containment for TriAL and several fragments of GXPath, are presented for the first time.

Remark 1. *Following the usual assumption of XML* data trees*, where each node carries a single data value [Bojanczyk et al., 2009, Kaminski and Tan, 2008, Segoufin, 2006], we will also consider graphs where each node has only one data value attached to it. Note that this is not a real restriction, as multiple attributes can be modelled with additional outgoing edges, labelled with the attribute name, and ending in a node whose data value is the value of the appropriate attribute. Furthermore, as we show in Section 2.1, one can go from one model to the other without having any effect on the presented results. There we also show how the graph from Figure 1.1 can be modelled using this assumption.*

This simplification is done mostly for the ease of notation, but, as already mentioned, all of the results still hold if one assumes nodes with multiple attributes.

1.3 Other related work

As we mentioned earlier most current approaches to querying graph database separate the data aspect and the topological aspect of such databases. That mixing of these two modes of querying is needed became apparent in the early days of graph database systems when users started asking questions about propagating the data along paths and patterns. The first system that recognized the necessity of treating both data and topology as equal was GOOD [Gyssens et al., 1994], however, the navigational features used there were rather rudimentary [Van den Bussche and Vossen, 1993], as the system was focused on managing object-oriented databases. The Lorel system [Abiteboul et al., 1997] partially addresses this problem by allowing conjunctive RPQs with variables returning nodes whose data values can be accessed and compared. This, however, still does not resolve the issue, as it does not allow data to be propagated along the path: it first extracts nodes using navigational queries (namely CRPQs) and after that filters the data from extracted nodes by a relational mechanism. Interestingly, despite these deficiencies, the system actually matches many capabilities of current commercial graph systems

such as Neo4j [Neo4j, 2013], Dex [Dex, 2013], or Gremlin [Gremlin, 2013]. Several other systems based on similar principles were developed in 1990s and 2000s ([Fernández et al., 2000, Amer-Yahia et al., 2009] – for a survey see [Angles and Gutierrez, 2008]), but to the best of our knowledge none of them had the ability to ask queries that mix data and topology beyond basic tasks that essentially amount to treating the two separately. Furthermore, the main concern in these approaches was usability and they were seldom looked at from a theoretical perspective, so issues such as query evaluation and static analysis aspects of these languages are not that well understood.

Chapter 2

Preliminaries

In this chapter we will provide necessary background information about graph databases, formally define the model used throughout this thesis and give a brief overview of graph query languages studied in the past. We begin by describing the model in Section 2.1 and explaining how it generalizes the usual graph data model. We also illustrate how theoretical restrictions imposed by the formal definition can easily be lifted in a more applied setting which requires multiple attributes and values per node, as in e.g. social networks. Following that we define the class of regular path queries, or RPQs, which had formed the basis of every graph database language since its inception in the late eighties [Cruz et al., 1987, Consens and Mendelzon, 1990]. Following that we will review information about some more general languages recently proposed in the context of RDF databases [Pérez et al., 2010].

One of the main issues governing the design of a query language is the efficiency of the query evaluation problem. Indeed, it is this problem that often makes or breaks a proposed language and some elegant theoretical constructions have to be discarded if they give an unreasonable rise in computational complexity of this problem. In Section 2.4 we define the query evaluation problem formally and review main results about classical graph query languages such as RPQs and NPQs.

Lastly, in Section 2.5 we discuss differences and similarities between two main language design principles for graph databases. Namely, we identify classes of path queries, whose main design principle is to define sets of permissible paths using some language theoretic formalism, and graph queries, that operate directly on graph, usually going beyond the reach of paths. We also show how path queries can be redefined to work directly on graphs and show that the two approaches are equivalent.

2.1 Graph databases

As mentioned in the introduction, the model of data we consider here is that of a graph database. In what follows we will take the approach where data resides in the nodes, however a different approach, with data residing in the edges is also possible and later on we will show that the two are equivalent. Next we define graph databases formally.

Let Σ be a finite alphabet, and $\mathcal{D}$ a countably infinite set of data values. Data graphs will have edges labelled by letters from Σ and nodes that store data values from $\mathcal{D}$.

Definition 2.1.1 (Data graphs)**.** *A* data graph, *or a* graph database *(over Σ and $\mathcal{D}$) is a triple $G = \langle V, E, \rho \rangle$, where:*

- *V is a finite set of nodes;*
- *$E \subseteq V \times \Sigma \times V$ is a set of labeled edges; and*
- *$\rho : V \to \mathcal{D}$ is a function that assigns a data value to each node in V.*

An example of a graph database is given in Figure 2.1. Here we assume that edge labels are a, b and data values are integers.

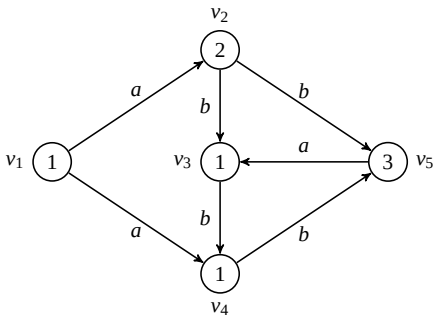


Figure 2.1: A graph database with data values

Note that traditionally [Cruz et al., 1987, Angles and Gutierrez, 2008, Calvanese et al., 2003] graph databases had no data values attached to them and thus amounted to finite edge labelled graphs. When we disregard data values and consider only edge labels we simply drop the function ρ from the above definition.

Query languages that do not refer to data values, but only traverse graph edges, such as RPQs and NRQs introduced below, will be called **navigational languages**.

On single value vs. multiple values Here we assume that each node has only a single data value assigned to it. In a more applied setting, such as the one presented in Figure 1.1, we might want to view nodes as small databases themselves, thus storing multiple data values or relations. Assumption that each node has only a single data value is not a real restriction

as multiple attributes can be modelled by extra outgoing edges from one node, each with the attribute name as the label and attribute value as the data value of the node it points to. This solution is illustrated in Figure 2.2. Furthermore, the way we design languages will make it easy to extend them to work with multiple data values.

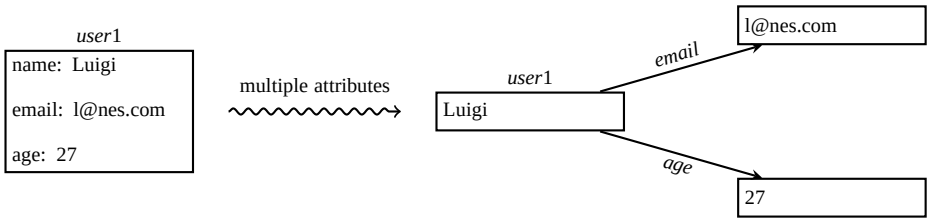


Figure 2.2: Dealing with multiple data values in a social network

Applying such a transformation to the graph in Figure 1.1, we would obtain the following graph. Note that for compactness of presentation we only show how to model the age attribute of certain actors, since this is all we will need in the future examples.

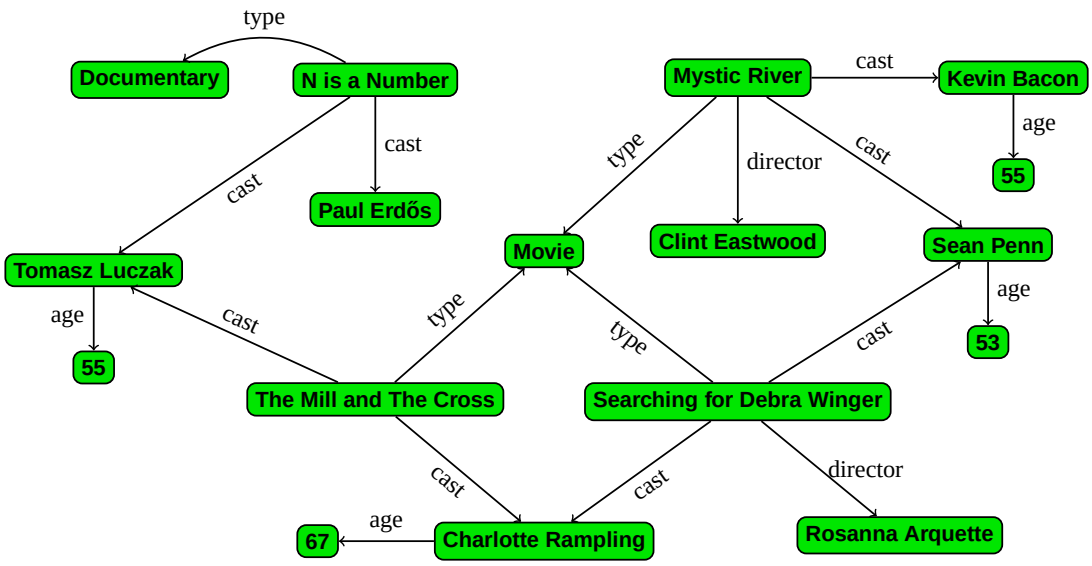


Figure 2.3: A movie database represented as a graph – now with a single data value per node

Placement of labels and data values In defining our model we followed the traditional approach where labels reside on the edges and data values in the nodes [Abiteboul et al., 1999, Cruz et al., 1987, Mendelzon and Wood, 1995, Consens and Mendelzon, 1990]. Other approaches are of course possible and have been considered over the years. For example in the XML setting it is usual that labels as well as data values are attached to the nodes, while child edges in trees modelling the data remain unlabelled [Neven, 2002, Segoufin, 2007, Figueira, 2010a, Figueira, 2010b]. Regarding data values, it may also make sense to place them on the

edges, for example when each edge label also has an associated value attached to it, as in e.g. [Ioannidis et al., 2011]. And there is, of course, the approach where both edges and nodes carry labels and data [Neo4j, 2013, Dex, 2013]. All of these approaches have their pros and cons, however it is easy to see that they are all essentially equivalent. Since the setting we will be using is fixed (that is, we assume labels on edges and values in nodes), all of the query languages will be designed to operate in this setting. However, it is important to note that this poses no restrictions, as all of the languages can easily be redefined to accommodate for data values in the edges, or labels in the nodes, without affecting any of the complexity bounds. To see this assume that we have a model with both nodes and edges carrying a label from a finite alphabet and a datum from an infinite domain. We could then assume that this model amounts to "splitting" edges into two and adding self loops to emulate node labels. This process is illustrated below.

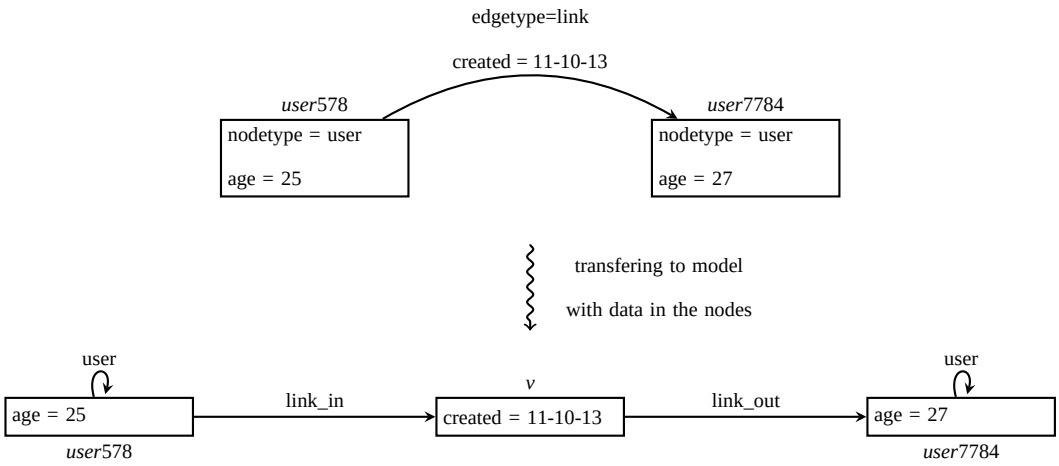


Figure 2.4: Simulating the model with data in both nodes and edges

Here we assumed that nodetype and edgetype attributes are simply labels from a finite alphabet, while age is an integer. Each edge is replaced by a node with one incoming and one outgoing edge corresponding to the original edge label, pointing to and from a node with the data value. Labels on the nodes are simulated by self loops. This shows how we can view graphs with data both in the nodes and in the edges as data graphs from our definition. It is important to note that, in the case when data is stored assuming values both in nodes and edges, one does not need to restructure the data, as queries can be modified in run-time, taking into account the way the data is stored.

On node ids and data values It is important to remark here that data values do not amount to node ids. Indeed, in the database from Figure 2.1 both nodes v_1 and e.g. v_4 have the same data value, namely 1, but they are not the same entity in the database. This illustrates that in general

data values can not be used as node ids, unless we assume that each node is assigned a different data value. For reasons discussed in Section 3.2, none of the query languages considered in this thesis will allow checking this, thus making it a global statement outside of the reach of our model. Furthermore, assuming that node ids are the same as data values might lead to some confusion in e.g. a genealogy database where two nodes might carry the same name, but it is important to be aware that they represent a different entity.

Paths Most of the classical graph query languages rely on defining paths between two nodes of a graph. In graphs with data, paths, however, carry some extra information. Consider, for example, a path $v_1v_2v_5v_3$ in the graph from Figure 2.1. If we traverse it by starting in v_1, reading its data value, then reading the label of (v_1, v_2), then the data value in v_2, etc., we end up with the following sequence: $1a2b3a1$. We shall refer to such sequences as *data paths*.

Next we define the notion of paths and data paths formally.

A path between nodes v_1 and v_n in a graph is a sequence

$$\pi = v_1 a_1 v_2 a_2 v_3 \dots v_{n-1} a_{n-1} v_n \tag{2.1}$$

such that each (v_i, a_i, v_{i+1}), for $i < n$, is an edge in E. Corresponding to the path π (2.1) we have a *data path*

$$w_\pi = \rho(v_1) a_1 \rho(v_2) a_2 \rho(v_3) \dots \rho(v_{n-1}) a_{n-1} \rho(v_n) \tag{2.2}$$

which is a sequence of alternating data values and labels, starting and ending with data values. The set of all *data paths*, i.e., such alternating sequences over Σ and $\mathcal{D}$, will be denoted by $\Sigma[\mathcal{D}]^*$. For both paths and data paths, we use the notation $\lambda(\pi)$ or $\lambda(w_\pi)$ to denote their label, i.e. the word $a_1 \dots a_{n-1} \in \Sigma^*$.

2.2 Regular path queries and extensions

The core class of queries for graph databases is the one of regular path queries or RPQs. These queries are purely navigational and disregard data values. However, as we will shortly see, they form a natural base for all languages that include any sort of navigation in graphs. RPQs are based on the principle of describing permitted paths in a graph. Since edges in data graphs are labelled by letters from a finite alphabet it is natural to describe the set of permitted paths as a regular language over this alphabet.

Regular path queries Formally *regular path queries*, or RPQs for short, are queries of the form $Q = x \xrightarrow{L} y$, where L is a regular language over some fixed finite alphabet Σ, specified by a regular expression or a finite state automaton [Cruz et al., 1987, Consens and Mendelzon, 1990, Calvanese et al., 2003]. Given a data graph G (the data in the nodes will be irrelevant for

RPQs), answer of a query Q on G, denoted by $Q(G)$, is the set of all pairs (v,v') of nodes in G such that:

- There is a path π in G, starting with v and ending with v', and
- The label $\lambda(\pi)$ is a word from L.

Note here a degree of separation between queries and language formalisms defining them. Namely, we have a regular expression (or an NFA) defining the language L of permissible paths (or rather their labels), while the query Q itself looks for paths in a graph whose label belongs to this set L of permissible paths. We will call such languages **path languages** since they amount to finding a path in the graph and matching the label of this path with a corresponding language defining the query.

An example of an RPQ is e.g.

$$Q = x \xrightarrow{(ab)^*} y.$$

From database in Figure 2.1 this query will extract e.g. (v_1,v_4) since path $\pi = v_1 a v_2 b v_5 a v_3 b v_4$ has the label *abab* which belongs to the language of $(ab)^*$. The other pairs in the answer $Q(G)$ are $(v_1,v_5),(v_1,v_3)$ and (v_5,v_4).

The fact that regular languages are closed under conjunction can lead us to a conclusion that taking two regular expressions e_1 and e_2 one can define a query which extracts pairs of nodes connected by two path, one in e_1 and another in e_2. However the expression defining intersection of e_1 and e_2 specifies a query that returns nodes connected by a single path whose label belongs to both languages. In fact, to define queries asking for multiple paths one has to use conjunctive regular path queries (CRPQs).

Conjunctive regular path queries *Conjunctive RPQs*, or *CRPQs* [Consens and Mendelzon, 1990] are the closure of RPQs under conjunction and existential quantification. Formally, they are expressions of the form

$$\varphi(\overline{x}) \;=\; \exists \overline{y} \bigwedge_{i=1}^{n} (z_i \xrightarrow{L_i} u_i), \tag{2.3}$$

where all variables z_i, u_i come from $\overline{x},\overline{y}$. The semantics naturally extends the semantics of RPQs: $\varphi(\overline{a})$ is true in G iff there is a tuple $\overline{b}$ of nodes such that, for every $i \leq n$, every pair v_i, v_i' interpreting z_i and u_i is in the answer to the RPQ $z_i \xrightarrow{L_i} u_i$.

We can now ask queries as e.g. the following one:

$$\varphi(x,y) = (x \xrightarrow{b^*} y) \wedge (x \xrightarrow{ba} y).$$

The query φ will return all pairs (v,v') of nodes such that there are paths π_1 and π_2, both starting with v and ending with v' such that $\lambda(\pi_1)$ belongs to language of b^*, while $\lambda(\pi_2)$ equals *ba*. Applied to the graph in Figure 2.1 this query will return (v_2,v_3).

Two-way regular path queries A natural extension of RPQs is to allow them to traverse graph edges backwards. Indeed, such a functionality is often required in practical scenarios, for example in a genealogy database one might want to reason about ancestors and in a crime detection scenario links are often tracked backwards to locate the main supplier of trafficked goods. RPQs extended with this ability are called *two-way regular path queries* or 2RPQs [Consens and Mendelzon, 1990, Calvanese et al., 2000, Calvanese et al., 2003].

Formally, let Σ be a finite alphabet. We will denote by $\Sigma^{\pm}$ the set $\Sigma \cup \{a^- : a \in \Sigma\}$. The letter a^- denotes that an edge is supposed to be traversed in a backward direction (note that edge labels can also be viewed as binary relations between nodes, thus a^- would be the reverse of relation a). If $p \in \Sigma^{\pm}$ we use p^- to denote the inverse of p. That is if $p = a$, for some $a \in \Sigma$ then $p^- = a^-$, and if $p = a^-$, then $p^- = a$. A 2RPQ over Σ is then an expression of the form $Q = x \xrightarrow{e} y$, where e is a regular expression or a finite state automaton over $\Sigma^{\pm}$.

In order to define semantics of 2RPQs we need the notion of a semipath. A *semipath* between nodes v_0 and v_n in a graph $G = (V, E)$ is a sequence π of the form $v_0 p_1 v_1 p_2 \dots p_n v_n$, where $n \geq 0$ and for each i we have $p_i \in \Sigma^{\pm}$ and $(v_{i-1}, a, v_i) \in E$ if $p_i = a$ and $(v_i, a, v_{i-1}) \in E$, if $p_i = a^-$. Intuitively, a semipath amounts to traversing graph edges both backwards and forwards, as dictated by the sequence of labels $p_1, \dots, p_n$. Then the answer to a 2RPQ Q over G, denoted $Q(G)$, is the set of all pairs (v, v') of nodes connected by a semipath whose label $\lambda(\pi) = p_1 \cdots p_n$ belongs to the language of e.

A sample 2RPQ is e.g.

$$Q = x \xrightarrow{a^- b^- b^*} y.$$

For a graph in Figure 2.1 we have $Q(G) = \{(v_3, v_5), (v_3, v_2), (v_3, v_3), (v_3, v_4)\}$.

It is straightforward to define a class of conjunctive queries using 2RPQs as atoms, much like CRPQs use RPQs. This class of queries is called *conjunctive two-way regular path queries* or C2RPQs.

2.3 Nested regular expressions

One of the most apparent shortcomings of RPQs and related formalisms is their inability to abstract away from paths. In semi-structured data one often needs to define patterns connecting certain nodes, or exhibit some structural properties of the underlying model that can not be captured by paths alone. For example in a social network scenario we might want to test if there is a chain of users connected by friends links and that along this chain each person likes the same type of music. This would be modelled by checking for an outgoing edge labelled *likes* to a node representing some music type (here we assume that the number of types is known in advance; in a more realistic setting we will need data values to model types of music). Note that since the length of such a chain can be arbitrary this can not be defined using CRPQs,

since the number of conjuncts of a CRPQ is fixed in advance. Thus, even though they can define some simple patterns, CRPQs fail to express many properties of interest when querying graphs. Indeed, the importance of ability to define patterns instead of paths was recognized in the study of XML, where even the most basic languages allow branching from the main path and checking if a certain condition is satisfied along the path. XML languages, and most notably XPath [Xpath, 1999, Benedikt and Koch, 2008, ten Cate and Marx, 2007], considered to be the logical core for querying XML documents [ten Cate and Lutz, 2009], form a good basis for graph language design and in later chapters we will show how the underlying ideas can be transferred from XML to graphs.

The first language influenced by XPath's functionality to allow branching away from a path (and thus defining patterns) is that of nested path queries, or NPQs. This language, first introduced in [Pérez et al., 2010], was created in order to capture certain navigational aspect of RDF documents [Klyne and Carroll, 2004] that lie beyond reach of the proposed SPARQL standard [Harris and Seaborne, 2013]. The expressions defining NPQs, called nested regular expressions, are themselves quite simple and amount to extending RPQs with inverses and nesting operators. The intuition behind nesting is that it acts like a test that a certain node in the path has to satisfy. The test itself is defined by a nested regular expression – hence the name. Next we define NREs.

Nested regular expressions, or NRE, over a finite alphabet Σ extend ordinary regular expressions with the nesting operator and inverses [Pérez et al., 2010, Barceló et al., 2012c]. Formally they are defined as follows:

$$n := \varepsilon \mid a \mid a^- \mid n \cdot n \mid n^* \mid n + n \mid [n]$$

where a ranges over Σ.

Intuitively NREs define binary relations consisting of pairs of nodes connected by a path specified by the NRE. When interpreted on a data graph G the relations are defined inductively as follows:

$$\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\} \\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\} \\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\} \\
[\![n \cdot n']\!]^G &= [\![n]\!]^G \circ [\![n']\!]^G \\
[\![n + n']\!]^G &= [\![n]\!]^G \cup [\![n']\!]^G \\
[\![n^*]\!]^G &= \text{the reflexive transitive closure of } [\![n]\!]^G \\
[\![[n]]\!]^G &= \{(v,v) \mid \exists v' \text{ such that } (v,v') \in [\![n]\!]^G\}.
\end{aligned}$$

A nested path query, or NPQ, is an expression of the form $Q = x \xrightarrow{e} y$, where e is a NRE. Given a data graph G, the answer to Q on G, denoted $Q(G)$ is the set $[\![e]\!]^G$.

An example of an NPQ is the e.g.:

$$Q = x \xrightarrow{(b[a^-])^+} y.$$

It checks that node at the end of each b-labelled edge also has an incoming a-labelled edge. For the graph in Figure 2.1 we have $[\![e]\!]^G = \{(v_2, v_3), (v_2, v_4), (v_3, v_4)\}$. Note that (v_2, v_5) is not in the answer to e since v_5 has no incoming a-labelled edges.

Note that the semantics of a NPQs is defined directly on graphs, not taking a detour through language theory like e.g. RPQs do. We will call such languages **graph languages**.

2.4 Query evaluation

One of the main problems associated with query languages is that of *query evaluation*, or as it is sometimes called, *query answering*. Indeed, gauging applicability of some language often depends on obtaining desirable complexity bounds of this problem. Studying query evaluation problem for a wide range of graph query languages that deal with data values constitutes the main portion of this dissertation and throughout the subsequent chapters we will explore how different features impact the complexity of the problem.

To define the query evaluation problem formally assume that we have a query language $\mathcal{L}$ over some finite alphabet Σ and a query $Q(\overline{x})$ from $\mathcal{L}$ returning tuples of nodes from a data graph G. Here we write $Q(\overline{x})$ to denote that Q returns tuples of length $|\overline{x}|$. The query evaluation problem for language $\mathcal{L}$ is then defined as follows:

PROBLEM:	QUERY EVALUATION($\mathcal{L}$)
INPUT:	A query $Q(\overline{x})$ with $\|\overline{x}\| = k$, a graph database G over Σ and a tuple $\overline{v} \in V^k$.
QUESTION:	Is $\overline{v} \in Q(G)$?

When studying query evaluation we will be interested in the complexity of this problem. Stated as above, this is often referred to as **combined complexity** of query evaluation problem [Vardi, 1982]. In databases we are often interested in the variant of this problem where the query Q is fixed, and only the graph G (together with tuple $\overline{v}$) is given as the input. This version is referred to as the **data complexity** of the query evaluation problem.

We will now review basic results about combined and data complexity of the languages introduced in previous sections.

Fact 2.4.1 ([Cruz et al., 1987])**.** *Both data and combined complexity of evaluating RPQ queries are* NLOGSPACE*-complete.*

This easily follows from the observation that in the case of RPQs one is given a graph G and a tuple of nodes s,t, along with the regular expression e as the input. To check if $(s,t) \in Q(G)$,

where $Q = x \xrightarrow{e} y$, it suffices to observe that G can be viewed as an automaton with s the initial and t the final state. Then the result follows from performing classical product construction of the graph with the automaton for e, where we check this product for nonemptiness on-the-fly. The lower bound follows from the fact that complexity of reachability in graphs is NLOGSPACE-hard [Jones, 1975].

It was also shown that if one allows only simple paths in a graph (that is paths that repeat no nodes), then both data and combined complexity jump to NP-complete [Mendelzon and Wood, 1995]. We however do not require paths to be simple, so the mentioned result does not affect our presentation.

When moving to CRPQs a jump in combined complexity occurs.

Fact 2.4.2 ([Consens and Mendelzon, 1990, Barceló et al., 2012b])**.** *Combined complexity of evaluating CRPQs is* NP*-complete. Data complexity is* NLOGSPACE*-complete.*

The data complexity bound follows from the same technique as for RPQs (but now using multiple automata). Bound for combined complexity is obtained by guessing a polynomial length witnessing paths and verifying that the guess is correct. The lower bound follows from a matching bound for relational conjunctive queries [Chandra and Merlin, 1977].

It is also known that adding inverses incurs no extra computational cost.

Fact 2.4.3 ([Calvanese et al., 2000])**.** *Both combined and data complexity of evaluating 2RPQs are* NLOGSPACE*-complete.*

This observation is straightforward, since evaluating 2RPQs is the same as evaluating RPQs over an extended alphabet.

For NPQs query evaluation is very efficient. In fact it is linear.

Fact 2.4.4 ([Pérez et al., 2010])**.** *Both combined and data complexity of evaluating NPQs are in* PTIME*. In fact, checking if a pair (v,v') belongs to $Q(G)$ can be done in $O(|G| \times |e|)$, where $Q = x \xrightarrow{e} y$.*

This algorithm relies heavily on the solution to the model checking problem for propositional dynamic logic [Harel et al., 2000].

2.5 Path languages and Graph languages

Examining carefully the semantics of NPQs one can see that they are in fact defined to operate directly on graphs, without taking an intermediate step through language theory as e.g. RPQs do. Indeed, the distinction between NREs and NPQs is purely artificial, and introduced only in order to keep the notation consistent throughout the thesis. We have already mentioned that such languages, whose semantics is dependant on the graphs and not language theory

formalism defining the set of allowed paths, will be called graph languages and their semantics graph semantics.

RPQs on the other hand start with the premise of specifying the set of allowed path labels and then their semantics is defined by finding paths in the graph whose label belongs to this set of allowed paths. Therefore there is a certain duality when dealing with such languages, which we call path languages. Namely, there is a language theoretic formalism (regular languages in the case of RPQs) that defines the set of allowed path labels and then there is the query itself whose semantics depends on two things:

1. Finding paths in the graph, and
2. Checking that the path label belongs to the language of the defining expression.

We have mentioned already that such languages are called path languages, since they rely on finding paths in the graph and do not operate on the graphs themselves. In order to underline this connection between queries and language theoretic models defining them we will be using such a duality between expressions defining path labels and the queries themselves, when appropriate. Therefore in the forthcoming chapters we will be dealing with:

- **Path languages** – when the underlying idea is to describe the set of permissible path labels and then the semantics calls for finding paths in the graph whose labels belong to this set.

- **Graph languages** – when queries are defined to operate directly on graphs and when paths alone no longer suffice to capture the intended semantics.

Important thing to note is that e.g. NREs can not be used in the same manner as regular expressions, since they no longer define paths, but patterns. Indeed, using the nesting operator, one can specify various patterns in a graph that are no longer captured by paths alone. Note that NREs could also be used to define sets of words (i.e. their semantics could be adopted to paths instead of graphs), where the nesting would only look ahead (or backwards) along a single path; however, this approach, although interesting in its own right [Reutter, 2013a], falls outside the scope of this thesis.

An important and useful observation is that path languages can always be defined to operate directly over graphs, where the definition simply captures the intended behaviour of navigating the graph along a path with the permissible label. This is particularly useful when one wants to define the semantics of e.g. the inverse operator, since the somewhat counter intuitive notion of a semi-path is no longer needed. In fact defining semantics of path queries directly on graphs, called the *graph semantics of path queries*, also gives a uniform way of looking at queries that is in a sense more relational then the traditional path semantics given above. However, due to

historical reasons, and to exemplify the underlying design principle of path queries, we will in general use the path semantics when dealing with such queries.

Next we show how to define graph semantics for RPQs.

Graph semantics for RPQs Here we define graph semantics of RPQs and 2RPQs formally and show that it matches the path semantics above. Recall that 2RPQs (which subsume the class of RPQs) are defined using expressions specified by the following grammar:

$$e := \varepsilon \mid a \mid a^- \mid e \cdot e \mid e^* \mid e + e, \tag{2.4}$$

where a ranges over a fixed finite alphabet Σ. Note that these are simply regular expressions over the extended alphabet $\Sigma^\pm$, just as in the definition of 2RPQs.

The graph semantics of such an expression e over a graph database G is then defined as follows:

$$\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\} \\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\} \\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\} \\
[\![e \cdot e']\!]^G &= [\![e]\!]^G \circ [\![e']\!]^G \\
[\![e + e']\!]^G &= [\![e]\!]^G \cup [\![e']\!]^G \\
[\![e^*]\!]^G &= \text{the reflexive transitive closure of } [\![e]\!]^G
\end{aligned}$$

In the end we simply define (2)RPQs as queries of the form $Q = x \xrightarrow{e} y$, where e is defined by the grammar above, and set $Q(G) = [\![e]\!]^G$. Same as for NPQs and NREs, this extra step, separating expressions from the queries they define, is simply syntactic and we do so only to keep the notation uniform. Note that (2)RPQs now operate directly over graphs. It is however easy to show that the two semantics coincide.

Lemma 2.5.1. *Let e be an expression defined by the grammar 2.4. Then for any data graph G and a pair v, v' of nodes in G it holds that $(v,v') \in [\![e]\!]^G$ if and only if there is a semi-path π connecting v and v' such that the label $\lambda(\pi)$ belongs to the language of e, when e is viewed as a regular expression over $\Sigma^\pm$.*

Note here that when only RPQs are considered semi-paths are replaced by paths. The lemma is proved by a straightforward induction on the structure of the expression e.

Remark 2. *Since for graph semantics there is no longer a real difference between the expressions defining the queries and the queries themselves, we will often simply use the expressions to denote queries and vice versa. Therefore, we will use NREs when talking about NPQs, or use the expressions from grammar 2.4 when talking about 2RPQs.*

A short note on the structure Seeing how there is a divide amongst the class of navigational graph languages, it is only natural that in our search for suitable querying framework for graphs with data we follow that divide. In that respect, we will begin our study using the more traditional approach of path languages in Part I, where various formalisms defining languages that handle data values will be used to describe the set of allowed paths. Here we will begin with some well established language theoretic formalisms, but will also define new ones, opening space to study them in separation, as well as when used to query graphs. Following that we will expand on the idea of NPQs and define several languages designed to work directly on graphs in Part II. There we will also consider what happens when we try to transfer ideas from graphs to a more general setting of RDF triplestores. Finally, in Part III we will examine how path and graph languages compare to each other, thus giving us a complete picture of the current landscape of languages for graphs with data.

Part I

Path languages

Chapter 3

From words to paths

In order to define queries on graphs with data we will have to decide whether we will be using the traditional approach of path queries (e.g. RPQs, 2RPQs), or the more general approach of graph queries such as NPQs. In this part of the dissertation we will concentrate on path queries, showing how, even when we want to reason not only about the shape of the path, but also about the values appearing along it, these can be defined using some standard language theoretic formalisms that take data value comparisons into consideration. In order to illustrate what a suitable formalism for describing both navigational and data aspects of graphs might be consider the following data graph.

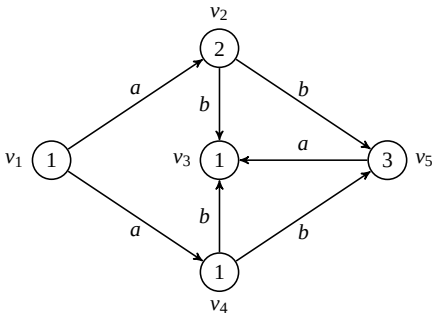


Figure 3.1: Graph database with data values

Over such a graph a typical RPQ may ask for pairs of nodes connected by a path from the regular language $(ab)^*$. In the graph in Fig. 3.1, one possible answer is (v_1, v_3), another – (v_1, v_5). To combine this with data values, we may ask queries of the following kind:

- Find nodes connected by a path from $(ab)^*$ such that the data values at the beginning and at the end of the path are the same. In this case, (v_1, v_3) is still in the answer but (v_1, v_5) is not.
- We may extend comparisons to other nodes on the path, not only to the first and the last

node. For example, we may ask for nodes connected by paths along which the data value remains the same, or on which all data values are different from the first one. The pair (v_1, v_3) is in the answer to the first query (the path $v_1v_4v_3$ witnesses it), while the pair (v_1, v_5) is in the answer to the second, as witnessed by the path $v_1v_2v_5$.

What kind of languages can we use in place of regular languages to specify paths with data? To answer this, consider, for example, a path $v_1v_2v_5v_3$ in the graph. If we traverse it by starting in v_1, reading its data value, then reading the label of (v_1, v_2), then the data value in v_2, etc., we end up with the following data path: $1a2b3a1$. Data paths are extremely close to an object that has been actively studied in the XML context – namely, *data words* [Bojanczyk, 2010, Bojanczyk et al., 2011, Segoufin, 2006, Segoufin, 2007]. A data word is a word in which every position is labelled by both a letter from a finite alphabet (e.g., a or b) and a data value (e.g., a number). Data paths are essentially data words with an extra data value. We can represent the data path $1a2b3a1$ as a data word $\binom{\#}{1}\binom{a}{2}\binom{b}{3}\binom{a}{1}$, where # is a special symbol reserved for the extra data value.

We can thus use multiple formalisms developed for data words (with a minor adjustment for the extra value) to specify data paths. Such formalisms abound in the literature, and include first-order and monadic second-order logic with data comparisons [Bojanczyk et al., 2009, Bojanczyk et al., 2011], LTL with freeze quantifiers [Demri and Lazić, 2009], XPath fragments [Bojanczyk, 2010, Figueira, 2009], and various automata models such as pebble and register automata [Bouyer et al., 2001, Kaminski and Francez, 1994, Kaminski and Tan, 2008, Kaminski and Tan, 2006, Neven et al., 2004].

The question is then, which one to choose? To answer this, we look at data complexity of query answering for each of these formalisms. We show that as long as the formalism is capable of expressing what is perhaps the most primitive language with data value comparisons (two data values are equal) and is closed under complementation, then *data* complexity is NP-hard. Clearly one cannot tolerate such high data complexity, and this rules out most of the above mentioned formalisms except *register automata*.

Before examining this issue, in the following section we will show how to go from data paths to data words and vice versa. In particular we will argue that the approach when graph databases are defined in such a way that data values reside in the nodes (as in Section 2.1) naturally gives rise to data paths, while graphs with data in the edges are better suited for working with data words. Both of the approaches have their strengths and weaknesses, but as we will shortly see, they are essentially equivalent.

3.1 Data words vs data paths

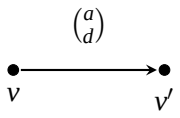
As mentioned before, data words can easily be used in place of data paths. To see this, consider e.g. a data path $1a3c1$. This data path can be replaced by the data word $\binom{\#}{1}\binom{a}{3}\binom{c}{1}$. Here we take the approach that the missing symbol from the finite alphabet is replaced by the special label #. Then when defining the language one has to make sure that the first letter symbol is not considered. This, however, will be easily achievable in any of the data word formalisms discussed below.

On the other hand, to move from data words to data paths we will have to add an extra data value. Let $\perp$ be a new data value, not used in the domain of the considered language. Then the data word $\binom{b}{1}\binom{a}{3}\binom{c}{1}$ is replaced by the data path $\perp b1a3c1$; that is, we add this special symbol $\perp$ to the start of the path to denote the missing data value.

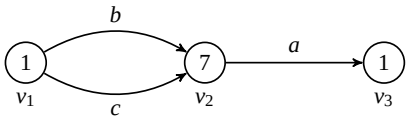
To see where this discrepancy between the two approaches comes from, consider a typical graph database, as for example the one in Figure 3.1. A path in this database is e.g. $v_1v_2v_5v_3$. When traversing this path we see that each edge comes with a label and two data values assigned to its ends. Therefore, by reading data values and edge labels in order in which they appear on this path we obtain the sequence $1a2b3a1$, that is, we end up with a data path. This approach, where data values are placed in the nodes is more usual for graph databases [Abiteboul et al., 1999] and has historically prevailed over the model where data values reside in the edges. One of the main reasons for this is the fact that in a graph database nodes are themselves considered to be small databases, thus carrying data, which is naturally modelled by data values from an infinite domain.

The dual approach, where data values reside in the edges, has by now been mostly abandoned. However, its main attraction is that it allows path labels to be described in terms of data words, which are, unlike data paths, symmetric objects, and thus much easier to manipulate. For example concatenating data words is straightforward, while doing the same for data paths requires some attention (namely, one has to make sure that the last value in the first path equals the first one in the second path). In what follows, mostly to stay with the traditional approach to graph querying, we will consider the model where data resides in the nodes, although, as we now show, the two approaches are equivalent. Note that this equivalence comes as a no surprise as a similar duality is present in the are of formal verification, where one can use both labelled transition systems and Kripke structures as models for temporal or modal logic formulas.

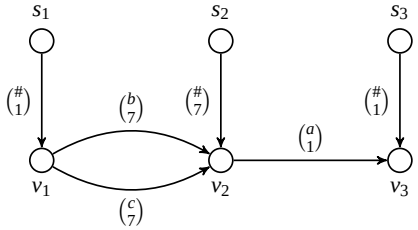
In a model where data values are in the edges a typical edge looks like the one in the following figure.



If we wanted to convert a usual data graph G, as defined in Section 2.1, we would have to, for each node v in G add a new node s_v and an edge labelled $\binom{\#}{\rho(v)}$ from s_v to v. Furthermore, each edge (v,a,v') in G has to be replaced by the edge $(v,\binom{a}{\rho(v')},v')$. This is illustrated in the following example:



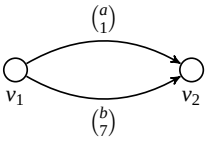
Graph G with data values in the nodes



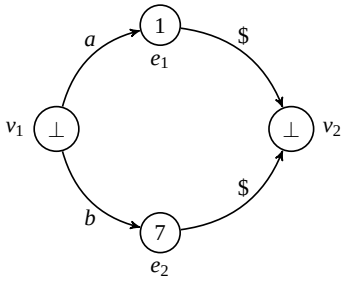
An equivalent graph G' with data values in the edges

To see that the two graphs from the figure above indeed represent the same set of data paths consider for example the path $\pi = v_1 b v_2 a v_3$ and the associated data path $1b7a1$. As we mentioned above we will represent this data path with the data word $\binom{\#}{1}\binom{b}{7}\binom{a}{1}$. But then the corresponding path in G' simply starts in s_1 and continues along the nodes from G, that is the whole path is $s_1\binom{\#}{1}v_1\binom{b}{7}v_2\binom{a}{1}v_3$ and the label of this path is obviously the one required. The intuition behind this transformation is to push data values to the incoming edge, with a new node s_v for every node v to allow it to be the start point of some path. Therefore we see that using data word formalisms to reason about data paths, or going from the model where data resides in the nodes to the one where it is in the edges, present no problems.

Going from graph with data values in the edges back to the ones where it is in the nodes is a bit more cumbersome, as now we can not simply push the value to one node, since there can be multiple edges between the nodes. The solution then, is to add a new node for each edge of the graph and assign it the data value of that edge. The new node is then connected to the graph by adding an extra label. All of the nodes from before are assigned the same data value $\perp$, signifying that this value should be skipped. This solution is illustrated in the following image.



An graph with data values in the edges



An equivalent graph with data values in the nodes

Note that here the equivalent data path would require a bit more padding than in the other case. For example the path $v_1\binom{a}{1}v_2$ would now correspond to $v_1 a e_1 \$ v_2$, and thus data path

$\perp a1\$\perp$, with special symbol \$ denoting that the following data value $\perp$ should be ignored. It is however easy to see that such a behaviour can easily be encoded by any of the data path formalisms we study in the following chapter.

Seeing how the two approaches differ, from now on we will use the traditional model where data resides in the nodes and develop language formalisms for describing data paths. As we have shown, it is straightforward to adapt data word formalisms to work in this setting, however, to keep notation consistent, we will redefine all of the data word formalisms to operate directly on data paths. We will briefly return to the setting of data words in Chapter 6, where we show how formalisms introduced specifically for data paths can be adapted to work on data words. In that chapter we will deal with main language theoretic issues connected to such languages and show how they relate one to another.

3.2 Ruling out bad alternatives

A **data path query** is an expression of the form $Q = x \xrightarrow{L} y$, where L is a set of data paths. Depending on which formalism we use to specify allowed languages L we will have different classes of data path queries.

Therefore, to talk about data path queries, as just defined, we need to express properties of paths with data. As we already mentioned, these are essentially data words, with an extra data value attached. Quite a few languages and automata models have been developed for data words over the past few years, mainly in connection with the study of XML, especially XPath. We now give a quick overview of them. A more extensive survey can be found in [Segoufin, 2006].

FO($\sim$) **and** MSO($\sim$) These are first-order logic and monadic second-order logic extended with the binary predicate $\sim$ saying that data values in two positions are the same. For example, $\exists x \exists y\ a(x) \wedge a(y) \wedge x \sim y$ says that there are two a-labeled positions with the same data value. Two-variable fragments of FO($\sim$) and existential MSO with the $\sim$ predicate have been shown to have decidable satisfiability problem [Bojanczyk et al., 2009, Bojanczyk et al., 2011].

Pebble automata These are basically finite state automata equipped with a finite set of pebbles. To ensure regular behavior pebbles are required to adhere to a stack discipline. The automata are modeled in such a way that the last placed pebble acts as the automaton head and we are allowed to drop and lift pebbles over the current position. In addition to this we can also compare the current data value to the one that already has a pebble placed over it. Algorithmic properties and connections with logics have been extensively studied in [Neven et al., 2004].

LTL$_\downarrow$ This is the standard LTL expanded with a freeze operator that allows us to store the current data value into a memory location and use it for future comparisons. The full logic has undecidable satisfiability problem, but various decidable restrictions are known [Demri and Lazić, 2009, Demri et al., 2007].

Register automata These are in essence finite state automata extended with a finite set of registers allowing us to store data values. Although first studied only on words over infinite alphabet [Kaminski and Francez, 1994, Neven et al., 2004, Sakamoto and Ikeda, 2000] they are easily extended to handle data words, as illustrated in [Demri and Lazić, 2009, Segoufin, 2006]. They act as usual finite state automata in the sense that they move from one position to another by reading the appropriate letter from the finite alphabet, but are also allowed to compare the current data value with ones already stored in the registers.

XPath fragments XPath is the standard language for navigating in XML documents, i.e., for describing paths in a way that may also include conditions on data values that occur in documents. Fragments of XPath (with and without data values) have been extensively studied, see, e.g., [Benedikt et al., 2008, Bojanczyk et al., 2009]. While in general the satisfiability problem is undecidable, several decidable restrictions are known, e.g., [Figueira, 2009, Figueira and Segoufin, 2011].

In deciding which formalism to choose, we look at the *data complexity* of evaluating data path queries, and try to rule out those for which data complexity is intractable. Technically, a formalism just defines a set of allowed languages $L \subseteq \Sigma[\mathcal{D}]^*$. As before, a query Q is then simply an expression of the form $Q = x \xrightarrow{L} y$. Thus each formalism for defining allowed languages L gives rise to an associated class of queries. It turns out that most of the formalisms for data words/paths are actually not suitable for graph querying. This is implied by the following result. Let L_{eq} be the language of data paths that contain two equal data values. We will denote its complement, i.e. the language of all data paths containing pairwise different data values by $\overline{L_{eq}}$.

Theorem 3.2.1. *The data complexity of evaluating $Q = x \xrightarrow{\overline{L_{eq}}} y$ over data graphs is* NP*-complete.*

Proof. The proof is by showing that with $\overline{L_{eq}}$, one can encode the 2-disjoint-paths problem which is NP-complete [Fortune et al., 1980]. This problem is to check, for a graph G and four nodes s_1, t_1, s_2, t_2 in G, whether there exist two paths in G, one from s_1 to t_1 and the other from s_2 to t_2 that have no nodes in common. First, we argue that we can assume that s_1, t_1, s_2, and t_2 to be distinct. This is because we can always add two new nodes for each repeated node

and connect them with all the nodes the repeated node was connected to, thus modifying our problem to have all source and target nodes different.

Assume that $G = \langle V, E \rangle$ is a digraph and s_1, t_1, s_2, t_2 are four distinct nodes in G. Recall that our query is $Q = x \xrightarrow{\overline{L_{eq}}} y$. Since the query will disregard edge labels we can take $\Sigma = \{a\}$. We will construct a data graph G' and two nodes $s, t \in G'$ such that $(s,t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2.

Let $V = \{v_1, \ldots, v_n\}$. The graph G' will contain two disjoint isomorphic copies of G (extended with data values and labels) connected by a single edge. We define the two isomorphic copies $G_1 = \langle V_1, E_1, \rho_1 \rangle$ and $G_2 = \langle V_2, E_2, \rho_2 \rangle$ by:

- $V_1 = \{v'_1, \ldots, v'_n\}$,
- $V_2 = \{v''_1, \ldots, v''_n\}$,
- $E_1 = \{(v'_i, a, v'_j) : (v_i, v_j) \in E\}$,
- $E_2 = \{(v''_i, a, v''_j) : (v_i, v_j) \in E\}$ and
- $\rho_1(v'_i) = \rho_2(v''_i) = i$, for $i = 1 \ldots n$,

and then let $G' = \langle V', E', \rho' \rangle$, where

- $V' = V_1 \cup V_2$,
- $E' = E_1 \cup E_2 \cup \{(t'_1, a, s''_2)\}$ and
- $\rho' = \rho_1 \cup \rho_2$.

Note that ρ' is well defined since V_1 and V_2 are disjoint. Finally, we define $s = s'_1$ and $t = t''_2$.

We claim that $(s,t) \in Q(G')$ if and only if there are two disjoint paths in G from s_1 to t_1 and from s_2 to t_2 in G. To see this assume first that $(s,t) \in Q(G')$. This means that we have a path in G' which starts in s'_1 and ends in t''_2. In particular, it must pass the edge from t'_1 to s''_2, since this is the only edge connecting the two graphs. Also, since all data values on this path are different, we know that no node can repeat, i.e., the path contains no two copies of the same node in G. But then we simply split this path into two disjoint paths in G since the structure of edges in G' is the same as the one in G with the exception of edge between t'_1 and s''_2.

Conversely, assume that we have two disjoint paths from s_1 to t_1 and from s_2 to t_2 in G. Notice that we can assume these two paths to contain no loops, since loops can be removed while keeping the paths disjoint. To obtain a data path from s to t in $\overline{L_{eq}}$, we simply follow the corresponding path from s'_1 to t'_1 in G_1 (and thus in G'), traverse the edge between t'_1 and s''_2 and then follow the path in G_2 (and thus in G') from s''_2 to t''_2 corresponding to the path from s_2 to t_2 in G. Since the two paths in G have no node in common and do not have loops, all data values on the constructed data path from s to t in G' are different.

This completes the proof. □

Note that L_{eq} is about the simplest property one can express about data paths/words; it would be hard to imagine a formalism that cannot check for the equality of data values. The corollary below effectively rules out closure under complement for such formalisms if they are to be used in graph querying.

Corollary 3.2.2. *Assume that we have a formalism for data paths that can define L_{eq} and that is closed under complement. Then data complexity of evaluating data path queries is* NP*-hard.*

This immediately rules out FO$(\sim)$ and its two-variable fragment, LTL with the freeze quantifier, and pebble automata.

The only hope we have among standard formalisms is *register automata*, since they are not closed under complementation [Kaminski and Francez, 1994]. In the following chapter we show that we can achieve good query answering complexity using register automata and some of their restrictions, while still retaining sufficient expressive power.

Remark 3. *It is important to note that we will come back to FO in Chapter 7, where its semantics will be defined directly on graphs. As a consequence, in that context negation will be limited to the active domain, and not to the set of all data words as here, therefore expressing that all data values along a path are different will no longer be possible.*

In Chapter 7 we will also come back to XPath, which we do not consider in the context of path queries. The main reason for this is the fact that XPath is intrinsically a graph (originally tree) language, and even when it is used to reason about data words the semantics relies on defining patterns [Figueira, 2010b] in a same way as on trees. Indeed, when used over data words XPath simply treats them as trees and is thus not a true path language. Another reason not to study XPath as a path language is that even the more general graph approach already yields very efficient query evaluation algorithms (combined complexity is always PTIME *and for some fragments even linear).*

Chapter 4

Languages for data paths

This chapter will consider classes of graph query languages based on the principle of defining paths in a graph. As already mentioned, we will take the classical approach of RPQs and consider language theoretic formalisms defining sets of data paths, while the query will then be satisfied if we can find a data path in the graph whose label belongs to the defined set. In that respect, we will differentiate between a language formalism used (e.g. regular expressions in the case of RPQs) and the class of queries they give rise to (that is RPQs).

In Chapter 3 we showed that due to unreasonably high data complexity most formalisms defining languages of data words(all of which can easily be adapted to define data paths) can be ruled out, with the notable exception of register automata.

These automata, originally introduced in [Kaminski and Francez, 1994] to work with words over infinite alphabets and later extended to data words [Segoufin, 2006, Demri and Lazić, 2009], give rise to a class of queries called regular data path queries, or RDPQs for short. Here we study their query answering problem and present an algorithm, based on computing the product of automata, which, when nonemptiness is checked on-the-fly, gives an NLOGSPACE data complexity and PSPACE combined complexity bound. The bound for data complexity is good (it matches the usual RPQs) and the bound for combined complexity is tolerable (equivalent to that of FO, but higher than the NP bound for conjunctive RPQs or the PTIME bound for RPQs).

However, automata are not an ideal way of specifying conditions in queries. In RPQs, we use regular expressions rather than NFAs. While some regular expressions have been considered for register automata [Kaminski and Tan, 2006], they are very far from intuitive[1] and lack the expressive power to capture register automata. Therefore we propose three types of regular expressions that can be used in queries, all of them subsumed by register automata.

The first, called regular expressions with memory and giving rise to regular queries with

[1] For instance to express the language L_{eq} of paths with two equal data values the formalism in [Kaminski and Tan, 2006] uses the expression $y^{*\{y\}} \cdot_\emptyset x \cdot_\emptyset y^{*\{y\}} \cdot_\emptyset x \cdot_\emptyset y^{*\{y\}}$, while the class of regular expressions with equality introduced in Section 4.4 defines the same language using a simple expression $\Sigma^* \cdot (\Sigma^+)_= \cdot \Sigma^*$.

memory (or RQMs), is close in spirit to automata themselves and it lets one store a data value and use it later. For example, to express the query "connected by a path along which the data value remains the same", we would use the expression $\downarrow x.(\Sigma[x^{=}])^*$. This expression says: store the first value of the path into x, and then go along, if labels are arbitrary (Σ) and the condition $x^{=}$, meaning that the value is equal to x, holds. These expressions are much easier to write than the automata, and at the same time they can be translated into register automata; thus data complexity of queries remains in NL. We show that the combined complexity remains the same as for automata, i.e., PSPACE-complete (except in a rather limited case when the Kleene star is not used: then it drops to NP-complete). Later on we will also show that they have the same expressive power as register automata.

One unusual feature of regular expressions with memory and the associated class of queries is that they do not define proper scope of variables. Indeed, the variable, once stored, can be used at any point further on. This behaviour, although necessary to show equivalence with register automata, seems very unnatural, so in the following section we study the language with proper scoping rules defined. We will show that this language is strictly weaker than the two above, however, this is not reflected on the evaluation problem, as it remains PSPACE-hard for combined complexity.

This motivates a third class of expressions that restrict the ability to compare data values along the path; instead, one can only do comparisons for chosen subexpressions. A simple example of such an expression is $\Sigma^{+}_{=}$, which denotes nonempty data paths that have same data value at the beginning and at the end of the path: Σ^{+} indicates the label of the path, and the subscript $=$ states the condition for the first and the last data values. A slightly more elaborate example is $\Sigma^* \cdot \Sigma^{+}_{=} \cdot \Sigma^*$. It says that a subpath conforms to $\Sigma^{+}_{=}$, i.e., it denotes data paths on which two data values are equal. For expressions of this kind, called regular expressions with equality, we give a polynomial-time algorithm for combined complexity. The key idea is to translate expressions into push-down automata and then take the product with an automaton obtained efficiently from the graph database.

Finally, we will consider variable automata, introduced recently in [Grumberg et al., 2010a] to define languages over an infinite alphabet. Here we redefine them on data paths and show that the corresponding class of queries, called regular queries with variables (or RQVs) has combined complexity of query evaluation between that of register automata and the much weaker regular expressions with equality. These automata themselves, however, are incomparable with register automata and can even not express some properties definable by regular expressions with equality.

4.1 Register automata as a query language

As stated in the previous chapter, register automata are the only standard formalism for defining classes of data words that does not immediately lead to NP-hard data complexity of queries on graphs with data. In this section we define them and study query evaluation for data path queries based on these automata. We will slightly alter the definition of register automata used in e.g. [Demri and Lazić, 2009, Segoufin, 2006] to work on data paths rather than data words, without affecting their desirable properties.

As mentioned earlier register automata move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to one previously stored into the registers. Our version of register automata will use slightly more involved comparisons which will be boolean combinations of atomic $=, \neq$ comparisons of data values.

To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then conditions in $\mathcal{C}_k$ are given by the grammar:

$$c \;:=\; x_i^{=} \mid x_i^{\neq} \mid e^{=} \mid e^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \quad 1 \leq i \leq k,$$

where e is a data value from $\mathcal{D}$, also referred to as the *constant*. Let $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$, where $\perp$ is a special symbol signifying that the register is empty. The satisfaction of a condition is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}_\perp^k$ as follows:

- $d, \tau \models x_i^{=}$ iff $d = d_i$;
- $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d, \tau \models e^{=}$ iff $d = e$;
- $d, \tau \models e^{\neq}$ iff $d \neq e$;
- $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1, \ldots, k\}$ and ε for a condition that is true for any valuation and data value (e.g. $c \vee \neg c$).

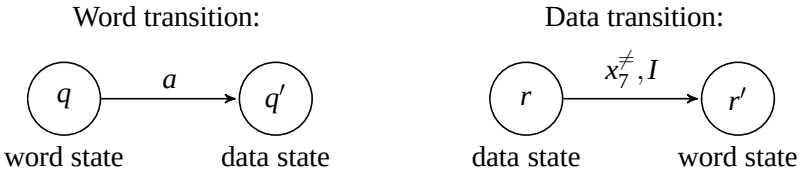
Definition 4.1.1 (Register data path automata)**.** *Let Σ be a finite alphabet, and k a natural number. A* k-register data path automaton *is a tuple $\mathcal{A} = (Q, q_0, F, \tau_0, \delta)$, where:*

- *$Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;*
- *$q_0 \in Q_d$ is the initial state;*
- *$F \subseteq Q_w$ is the set of final states;*
- *$\tau_0 \in \mathcal{D}_\perp^k$ is the initial configuration of the registers;*
- *$\delta = (\delta_w, \delta_d)$ is a pair of transition relations:*
  - *$\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;*
  - *$\delta_d \subseteq Q_d \times \mathcal{C}_k \times 2^{[k]} \times Q_w$ is the data transition relation.*

The intuition behind this definition is that since we alternate between data values and word

symbols in data paths, we also alternate between data states (which expect data value as the next symbol) and word states (which expect alphabet letters as the next symbol). We start with a data value, so q_0 is a data state, end with a data value, so final states, seen after reading that value, are word states.

In a word state the automaton behaves like the usual NFA (but moves to a data state using its word transition function). In a data state, the automaton checks if the current data value and the configuration of the registers satisfy a condition, and if they do, moves to a word state and updates some of the registers with the read data value. Both functionalities are illustrated in the following image, where in the data transition automaton checks if data value is different that the one stored in register seven and then moves to a word state while storing the value into registers from the set I.



Note that we could have modelled constants by storing them into the initial assignment (possibly using more registers). We put them into conditions however, to have a uniform way of handling them when we define RQBs and RQMs in the following sections. When the condition ε is used, or when $I = \emptyset$ (that is we do not store the data value into some register) we will omit them from the transition in the image above.

Now we formally define acceptance of a data path by a register automaton. Given a data path $w = d_0a_0d_1a_1 \ldots a_{n-1}d_n$, where each d_i is a data value and each a_l is a letter, a configuration of $\mathcal{A}$ on w is a tuple (j, q, τ), where j is the current position of the symbol in w that $\mathcal{A}$ reads, q is the current state and $\tau \in \mathcal{D}_\perp^k$ is the current content of the registers. The initial configuration is $(0, q_0, \tau_0)$ and any configuration (j, q, τ) with $q \in F$ is a final configuration.

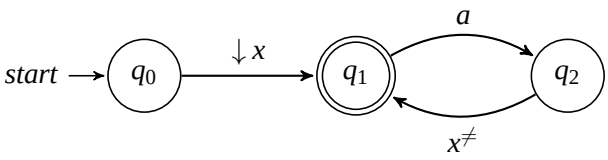
From a configuration $C = (j, q, \tau)$ we can move to a configuration $C' = (j+1, q', \tau')$ if one of the following holds:

- the jth symbol is a letter a, there is a transition $(q, a, q') \in \delta_w$, and $\tau' = \tau$; or
- the current symbol is a data value d, and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \tau \models c$ and τ' coincides with τ except that the ith component of τ' is set to d whenever $i \in I$.

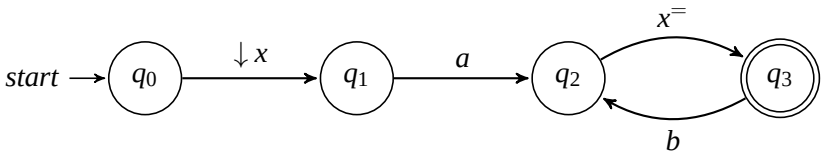
A data path w is accepted by $\mathcal{A}$ if $\mathcal{A}$ can move from the initial configuration to a final configuration after reading w. The language of data paths accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Example 4.1.2. *Next we provide three examples of data path languages and register automata recognizing them.*

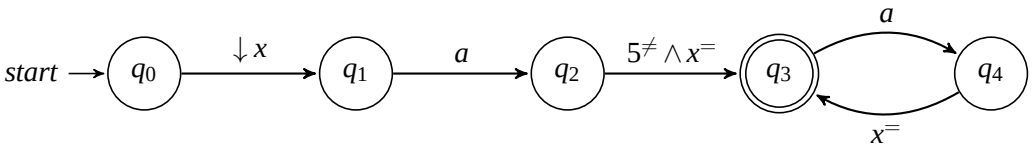
1. *The following automaton recognizes the language of all data paths where the first data value differs from all the others and the label is a^*. It operates by storing the first data value into the register x, which is denoted here and in the examples below by $\downarrow x$. It then moves to the state q_1, where it loops(by alternating between q_2 and q_1), while checking that the data value being read is different from the one stored in x. If this is satisfied it ends its computation in an accepting state q_1.*



2. *The language of paths where all data values are the same and the label of each path starts with an a and is then followed by an arbitrary number of bs is defined by the automaton below. Similarly as in the example above we store the first data value into the register x and then move to q_1, where the automaton checks that the first letter is a. It then proceeds to loop over bs by making sure that each data value equals to the one stored, ending its accepting run in the state q_3.*



3. *To illustrate how comparisons with constants work we now construct the automaton defining the language where each data value equals the first one, but the second value is different from 5. It proceeds as above, storing the first value into its register, with the exception that after reading the second value it explicitly checks if it is different from 5.*



Regular data path queries

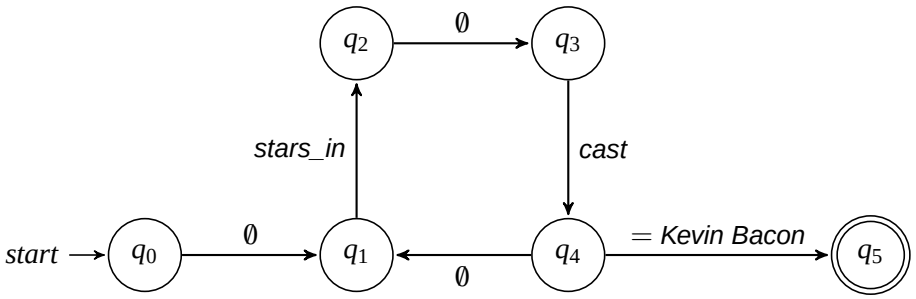
Our first class of queries on graphs with data is based on register data path automata.

Definition 4.1.3. *A* regular data path query (RDPQ) *over a fixed finite alphabet Σ is an expression $Q = x \xrightarrow{\mathcal{A}} y$ where $\mathcal{A}$ is a register data path automaton over Σ.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v,v') such that there is a data path w from v to v' that belongs to $L(\mathcal{A})$.

Example 4.1.4. *Coming back to the movie database from Figure 2.3, assume that, for each edge labelled* cast *that connects a movie or a documentary with an actor, we also have an edge going in the other direction labelled* stars_in*. For example we will add one such edge connecting* Kevin Bacon *with* Mystic River*, or* Charlotte Rampling *with* The Mill and The Cross*.*

We can then ask for all people who have a finite Bacon number using the query $Q = x \xrightarrow{\mathcal{A}} y$*, specified by the following register automaton* $\mathcal{A}$*:*



To improve readability we write $= c$ *instead of* $c^{=}$ *when comparing a data value with the constant* c*. The automaton works by traversing a sequence of* stars_in $\cdot$ cast *edges, which connect all pairs of actors who co-starred in a same film, but also makes sure that the last data value equals* Kevin Bacon*.*

Note that in addition to the actor with a finite Bacon number, this query also returns the node corresponding to Kevin Bacon.

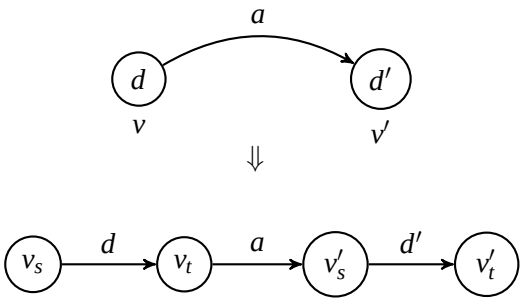
To evaluate RDPQs, we transform both a data graph G and a k-register data path automaton $\mathcal{A}$ into NFAs over an extended alphabet and reduce query evaluation to NFA nonemptiness. More precisely, to evaluate $Q(G)$, we do the following:

1. Let D be the set of all data values in G.

2. Transform $G = \langle V, E, \rho \rangle$ into a graph $G' = \langle V', E' \rangle$ over the alphabet $\Sigma \cup D$ as follows:

   - $V' = \{v_s, v_t \mid v \in V\}$
   - $E' = \{(v_t, a, v'_s) \mid (v, a, v') \in E\}$
     $\bigcup \{(v_s, \rho(v), v_t) \mid v \in V\}$

   Basically, we split each node v with a data value d into a source node v_s and a target node v_t and add a d-labeled edge between them; after that we restore the edges from E so that they go from target to source nodes. This is illustrated below.



3. Transform the automaton $\mathcal{A} = (Q, q_0, F, \tau_0, (\delta_w, \delta_d))$ into an NFA $\mathcal{A}_D = (Q', q_0', F', \delta')$ over the alphabet $\Sigma \cup D$ as follows:

   - $Q' = Q \times D_0^k$, with $D_0 = D \cup \{\perp\} \cup \{\tau_0(i) | i = 1 \dots k\}$;
   - $q_0' = (q_0, \tau_0)$;
   - $F' = F \times D_0^k$;
   - δ' includes two types of transitions.
     (a) Whenever we have a transition (q, a, q') in δ_w, we add transitions $((q,\tau), a, (q',\tau))$ to δ' for all $\tau \in D_0^k$.
     (b) Whenever we have a transition (q, c, I, q') in δ_d, we add transitions $((q,\tau), d, (q',\tau'))$ if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.

For two nodes v, v' of G, we turn G' into an NFA $\mathcal{A}_{G',v,v'}$ by letting v_s be its initial state and v_t' be its final state. Then we have the following.

Proposition 4.1.5. *Let $Q = x \xrightarrow{\mathcal{A}} y$ be an* RDPQ*, and G a data graph whose data values form a set $D \subseteq \mathcal{D}$. Then*

$$(v, v') \in Q(G) \quad \Leftrightarrow \quad L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset.$$

Proof. It follows immediately from the construction that the automaton $\mathcal{A}_D$ accepts precisely those data paths form $L(\mathcal{A})$ that have data values from D. To see this it suffices to show that every accepting run of $\mathcal{A}_D$ corresponds to an accepting run of $\mathcal{A}$ and vice versa, in the case of paths whose data values come form D. But this follows easily since $\mathcal{A}_D$ has all possible configurations of registers at it's disposal.

To see that the statement of Proposition holds assume first that $(v, v') \in Q(G)$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ from v to v' such that $w_\pi \in L(\mathcal{A})$. Since this is a data path in G starting with v and ending with v' it must also be a word in the language of $\mathcal{A}_{G',v,v'}$. On the other hand, since it is in $L(\mathcal{A})$, it must also be in $L(\mathcal{A}_D)$, since $\mathcal{A}_D$ is simply restriction of $\mathcal{A}$ to alphabet in which data values come only from the set D. Thus $L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset$.

Conversely, assume that $L(\mathcal{A}_{G',v,v'} \times \mathcal{A}_D) \neq \emptyset$. Then there is a data path $w_\pi = d_0 a_0 d_1 a_1 \dots a_{n-1} d_n$ such that $w_\pi \in L(\mathcal{A}_{G',v,v'})$ and $w_\pi \in L(\mathcal{A}_D)$. But then by construction w_π

must be a data path in G from v to v'. Also $w_\pi \in L(\mathcal{A})$, since $L(\mathcal{A}_D)$ is simply a restriction of language of $\mathcal{A}$ to data paths whose data values come from D. But this implies that $(v, v') \in Q(G)$. □

Thus, query evaluation, like in the case of the usual RPQs, is reduced to automata nonemptiness, although this time the automata are over larger alphabets. Since the construction is polynomial in the size of G and exponential in the size of $\mathcal{A}$ (as k gets into the exponent), we immediately get a PTIME upper bound for data complexity and an EXPTIME upper bound for combined complexity. By performing on-the-fly nonemptiness checking for the product, we can lower these bounds.

Theorem 4.1.6. *Data complexity of* RDPQ*s over data graphs is in* NL*, and the combined complexity of* RDPQ*s over data graphs is* PSPACE*-complete.*

We only need to prove PSPACE-hardness, since upper PSPACE bound follows from on-the-fly method for checking nonemptiness of exponential size automata. But this is an immediate consequence of Proposition 4.2.3 and Theorem 4.2.7, which are proved for a more restricted language.

The bound for data complexity cannot be lowered as there exist simple RPQs for which data complexity is NL-complete.

4.2 Regular queries with memory (RQMs)

Regular data path queries based on register automata have acceptable complexity bounds: data complexity is the same as for RPQs, and combined complexity, although exceeding the bounds on conjunctive queries and RPQs, is the same as for relational calculus or for RPQs extended with regular relations. Despite this, RDPQs as we defined them have no chance to lead to a practical language as it is inconceivable that the user will specify a register automaton over data paths. Even for queries such as RPQs and their extensions, conditions are normally specified via regular expressions.

Our goal now is to introduce regular expressions that can be used in place of register automata in data path queries. Note that as long as they express languages accepted by register automata, we shall achieve an NL bound on data complexity by Theorem 4.1.6.

The first class of queries, studied in this section, is based on an extension of regular expressions with *memory* that lets us specify when data values are remembered and when they are used. The basic idea is this: we can write expressions like $\downarrow x.a^+[x^=]$ saying: store the current data value in x and check that after reading a word from a^+ we see the same data value (condition $x^=$ is true). This will define data paths of the form $da \ldots ad$. Such expressions are

relatively easy to write and understand (much easier than automata), and the complexity of their query evaluation will not exceed that of register automata.

Definition 4.2.1 (Expressions with memory)**.** *Let Σ be a finite alphabet and $x_1,\dots,x_k$ a set of variables. Then* regular expressions with memory *are defined by the grammar:*

$$e \;:=\; \varepsilon \mid a \mid e+e \mid e\cdot e \mid e^+ \mid e[c] \mid \downarrow\overline{x}.e \tag{4.1}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1,\dots,x_k$.

A regular expression with memory e is well-formed if it satisfies two conditions:

- *Subexpressions e^+, $e[c]$, and $\downarrow\overline{x}.e$ are not allowed if e reduces to ε. Formally, e reduces to ε if it is ε, or it is e_1+e_2 or $e_1\cdot e_2$ or e_1^+ or $e_1[c]$ or $\downarrow\overline{x}.e_1$ where e_1 (and e_2) reduce to ε.*
- *No variable appears in a condition before it appears in $\downarrow\overline{x}$.*

The class of well-formed regular expressions with memory is denoted by $\mathrm{REG}(\Sigma[x_1,\dots,x_k])$.

The extra condition of being well-formed is to rule out pathological cases like $\varepsilon[c]$ for checking conditions over empty subexpressions, or $a[x^=]$ for checking equality with a variable that has not been defined. In what follows we always assume that regular expressions with memory are well-formed.

The intuition behind the expressions is that they process a data path in the same way that the register automaton would, by storing data values in variables, using these variables for comparisons and moving through the word by reading a letter from the finite alphabet. Note that when we bound a variable we do not specify the scope of this binding. This means that the variable can be used at any point after it was bounded till the end of the expression and is analogous to how register automata store and use data values.

Example 4.2.2. We now give four examples of such expressions and languages they recognize, before formally defining their semantics.

1. The expression $\downarrow x.(a[x^{\neq}])^+$ defines the language of data paths where all edges are labeled a and the first data value is different from all other data values. It starts by binding x to the first data value; then it proceeds checking that the letter is a and condition $x^{\neq}$ is satisfied, which is expressed by $a[x^{\neq}]$; the expression is then put in the scope of $+$ to indicate that the number of such values is arbitrary.
2. The expression $\downarrow x.(ab)^+[x^{\neq}]$ denotes the language of data paths whose label is of the form $ab\dots ab$ and for which the first data value is different from the last. Note that the

order of $+$ and condition is now different: the condition is checked after verifying that the label is in $(ab)^+$, i.e., at the end of the word.

3. The expression $\downarrow x.a^+[x^=] + \varepsilon$ denotes the language of data paths where all labels are a and the first data value is equal to the last. Note that one such data path is simply of the form d, for $d \in \mathcal{D}$, with label ε.
4. The language L_{eq} of data paths in which two data values are the same (see Section 3.2) is given by the expression $\Sigma^* \cdot \downarrow x.\Sigma^+[x^=] \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and Σ^* is the shorthand for $\Sigma^+ + \varepsilon$. It says: at some point, bind x, and then check that after one or more edges, we have the same data value.
5. The language where each data value equals the first one, but the second value is different from 5 is given by $\downarrow x.a[5^{\neq} \wedge x^=](a[x^=])^*$. It operates similarly as the expression in the first example, except that it tests for equality with the first data value, while explicitly testing that the second value differs from 5.

Semantics First, we define the *concatenation* of two data paths $w = d_1 a_1 \ldots a_{n-1} d_n$ and $w' = d_n a_n \ldots a_{m-1} d_m$ as $w \cdot w' = d_1 a_1 \ldots a_{n-1} d_n a_n \ldots a_{m-1} d_m$. Note that it is only defined if the last data value of w equals the first data value of w'. The definition naturally extends to concatenation of several data paths. If $w = w_1 \cdots w_l$, we shall refer to it as a *splitting* of a data path (into $w_1, \ldots, w_l$).

The semantics is defined by means of a relation $(e, w, \sigma) \vdash \sigma'$, where $e \in \mathrm{REG}(\Sigma[x_1, \ldots, x_k])$ is a regular expression with memory, w is a data path, and both σ and σ' are k-tuples over $\mathcal{D} \cup \{\perp\}$ (the symbol $\perp$ means that a register has not been assigned yet). The intuition is as follows: one can start with a memory configuration σ (i.e., values of $x_1, \ldots, x_k$) and parse w according to e in such a way that at the end the memory configuration is σ'. The language of e is then defined as

$$L(e) \;=\; \{w \mid (e, w, \overline{\perp}) \vdash \sigma \text{ for some } \sigma\},$$

where $\overline{\perp}$ is the tuple of k values $\perp$.

The relation $\vdash$ is defined inductively on the structure of expressions. Recall that the empty word corresponds to a data path with a single data value d (i.e., a single node in a data graph). We use the notation $\sigma_{\overline{x}=d}$ for the valuation obtained from σ by setting all the variables in $\overline{x}$ to d.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = d$ for some $d \in \mathcal{D}$ and $\sigma' = \sigma$.
- $(a, w, \sigma) \vdash \sigma'$ iff $w = d_1 a d_2$ and $\sigma' = \sigma$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff there is a splitting $w = w_1 \cdot w_2$ of w and a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.

- $(e^+, w, \sigma) \vdash \sigma'$ iff there are a splitting $w = w_1 \cdots w_m$ of w and valuations $\sigma = \sigma_0, \sigma_1, \ldots, \sigma_m = \sigma'$ such that $(w, w_i, \sigma_{i-1}) \vdash \sigma_i$ for all $i \in [m]$.
- $(\downarrow \overline{x}.e, w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma_{\overline{x}=d}) \vdash \sigma'$, where d is the first data value of w.
- $(e[c], w, \sigma) \vdash \sigma'$ iff $(e, w, \sigma) \vdash \sigma'$ and $\sigma', d \models c$, where d is the last data value of w.

Take note that in the last item we require that σ', and not σ, satisfies c. The reason for this is that our initial assignment might change before reaching the end of the expression and we want this change to be reflected when we check that condition c holds.

Translation into automata We now show that regular expressions with memory can be efficiently translated into register automata.

Proposition 4.2.3. *For each regular expression with memory $e \in \mathrm{REG}(\Sigma[x_1, \ldots, x_k])$ one can construct, in* DLOGSPACE*, a k-register data path automaton $\mathcal{A}_e$ such that $L(e) = L(\mathcal{A}_e)$.*

More precisely, the automaton $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$ (over data domain $\mathcal{D} \cup \{\bot\}$) has the property that for any two valuations σ, σ' and a data path w, we have $(e, w, \sigma) \vdash \sigma'$ iff the automaton $(Q, q_0, F, \sigma, \delta)$ has an accepting run on w that ends with the register configuration σ'.

Proof. We prove this by induction on the structure of e. Note that the initial assignment of $\mathcal{A}_e$ is not specified in advance. We will simply put the assignment in as needed, since it does not change the structure of the underlying automaton. In what follows we will identify the vector $\overline{x}$ of variables with the set of registers (i.e. positions) it corresponds to. For example the vector (x_3, x_5) will correspond to the set $I = \{3, 5\}$ of registers.

If $(e, w, \sigma) \vdash \sigma'$, we will write $w \in L(e, \sigma, \sigma')$ and similarly if $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$ started with σ accepts w with σ' in the registers, we write $w \in L(\mathcal{A}_e, \sigma, \sigma')$.

- If $e = \varepsilon$, then $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$, where $Q = \{d\} \cup \{w\}$ is the set of states, $q_0 = d$ is the initial state, $F = \{w\}$ the set of final states and the only transition is $(d, \varepsilon, \emptyset, w)$.
- If $e = a$, for some $a \in \Sigma$ then $\mathcal{A}_e = (Q, q_0, F, \overline{\bot}, \delta)$, where $Q = \{d_1, d_2\} \cup \{w_1, w_2\}$ is the set of states, $q_0 = d_1$ the initial state, $F = \{w_2\}$ the final state and the transition functions are as follows: $\delta_w = \{(w_1, a, d_2)\}$ is the word transition relation, and $\delta_d = \{(d_1, \varepsilon, \emptyset, w_1), (d_2, \varepsilon, \emptyset, w_2)\}$ is the data transition relation.
- If $e = e_1 + e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\bot}, \delta_1)$ and $\mathcal{A}_{e_2} = (Q_2, d_2, F_2, \overline{\bot}, \delta_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q, d_0, F, \overline{\bot}, \delta)$, where
  - $Q = Q_1 \cup Q_2 \cup \{d_0\}$, where d_0 is a new data state,

- $F = F_1 \cup F_2$,
- To δ we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(d,c,I,w) \in \delta_1 \cup \delta_2$, where $d = d_1$, or $d = d_2$, we add a transition (d_0,c,I,w).

To see that this automaton has the desired property assume that $w \in L(e_1 + e_2, \sigma, \sigma')$. This means $(e_1 + e_2, w, \sigma) \vdash \sigma'$. By definition, $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$. By the induction hypothesis it follows that either $\mathcal{A}_{e_1}$, or $\mathcal{A}_{e_2}$ accepts w and halts with σ' in the registers (when started with σ). From this it is clear that $\mathcal{A}_e$ can simulate the same accepting run when started with σ in the registers(by using the transition from d_0 to the appropriate automaton and continuing on the same run there). (Note that all conclusions here are equivalences.)

- If $e = \downarrow\overline{x}.e_1$ then again by the induction hypothesis we have $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\perp}, \delta_1)$ with the desired property. The automaton for $\mathcal{A}_e$ is defined as $\mathcal{A}_e = (Q_1 \cup \{d_0\}, d_0, F_1, \overline{\perp}, \delta)$, where d_0 is a new data state and δ contains all the transitions of $\mathcal{A}_{e_1}$ and in addition, for every transition (d_1,c,I,w), going from the initial state of $\mathcal{A}_{e_1}$, we add a transition $(d_0,c,I \cup \overline{x},w)$ to δ. The registers of $\mathcal{A}_e$ are the union of registers of $\mathcal{A}_{e_1}$ and $|\overline{x}|$ new registers.

  To see the equivalence, assume that $w \in L(e,\sigma,\sigma')$. By definition $(e,w,\sigma) \vdash \sigma'$. It follows that $(e_1, w, \sigma_{\overline{x}=v_1}) \vdash \sigma'$, where v_1 is the first data value in w and $\sigma_{\overline{x}=v_1}$ is the same as σ except that every register in $\overline{x}$ contains v_1. By the induction hypothesis we know that $\mathcal{A}_{e_1}$ with $\sigma_{\overline{x}=v_1}$ as initial assignment has an accepting run on w ending with σ' in the registers. But then $\mathcal{A}_e$ starting with σ in the registers can go through the same run with the exception that the first transition will change σ to $\sigma_{\overline{x}=v_1}$ and since all other transitions are the same we have the desired result. (Note that all conclusions here are equivalences.) It is important to note that potential confusion of the variables will cause no conflicts. To see this assume we have a transition (d_1,c,I,w) in $\mathcal{A}_{e_1}$ and we start with σ as initial assignment. If I and $\overline{x}$ have variables in common it will not matter, since all of them will get replaced by the same value, namely the first data value of w. This means that the first step of the run will end up with the same result. Also note that no transition in δ_d with d_1 as the first component will have $c \neq \varepsilon$, since this would amount to an expression starting with a condition, something disallowed by our syntax.

- If $e = e_1[c]$ then let $\mathcal{A}_{e_1} = (Q_1, d_1, F_1, \overline{\perp}, \delta_1)$ be an automaton for e_1 as before. We define $\mathcal{A}_e = (Q, d_1, F, \overline{\perp}, \delta)$ where $Q = Q_1 \cup \{w_f\}$, with w_f a new state, $F = \{w_f\}$ and for every transition (d,c',I,w) where $w \in F_1$ we add a transition $(d, c' \wedge c, I, w_f)$ to $\mathcal{A}_e$. We have to add a new state simply because our original automaton could have looped back from some final state.

  To get the equivalence assume again that $w \in L(e,\sigma,\sigma')$. By definition $(e_1,w,\sigma) \vdash \sigma'$ and

$\sigma', v \models c$, where v is the last data value in w. From the induction hypothesis we get an accepting run of $\mathcal{A}_{e_1}$ with σ as initial configuration and σ' as final one. But since $\sigma', v \models c$ instead of the last transition we can simply make a transition to w_f in $\mathcal{A}_e$ (since all other transitions are the same). We again notice that all the implications can be reversed, i.e. we can prove the equivalence.

- If $e = e_1 \cdot e_2$, take again $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ as above. The automaton for e is simply the union of the previous two automata, but in addition to the already existing transitions we add the following: for every (d, c, I, w) in $\mathcal{A}_{e_1}$, where $w \in F_1$ and for every (d_2, c', I', w') in $\mathcal{A}_{e_2}$, where d_2 is the initial state of $\mathcal{A}_{e_2}$, we add $(d, c \wedge c', I \cup I', w')$ to δ. Note that I is going to be an empty set, since we work with well formed expressions. We also make d_1 the initial state and F_2 the set of final states. The registers of $\mathcal{A}_e$ are again the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$.

  To get the desired result once again assume that $w \in L(e, \sigma, \sigma')$. This means $(e, w, \sigma) \vdash \sigma'$, which implies that there exists some σ'' and a splitting $w = w_1 \cdot w_2$ of w such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$. By the induction hypothesis we know that there is an accepting run of $\mathcal{A}_{e_1}$ on w_1 starting with σ and ending with σ'' in the registers and also an accepting run of $\mathcal{A}_{e_2}$ on w_2 starting with σ'' and ending with σ' in the registers. But we can simply combine these two runs into an accepting run of $\mathcal{A}_e$ on w. We do so by setting σ as initial assignment and tracing the run of $\mathcal{A}_{e_1}$ till the final state. Now instead of taking the last transition we will take one of the newly added transitions from the next to final state in $\mathcal{A}_{e_1}$ to the next to first state in $\mathcal{A}_{e_2}$. Note that we can do this since we know there is an accepting run of $\mathcal{A}_{e_2}$ on w_2 and since $w = w_1 \cdot w_2$, so their last and first data value, respectively, coincide. Note that at this point we end up with σ'' in the registers and can continue the accepting run of $\mathcal{A}_{e_2}$ and thus $\mathcal{A}_e$.

  Conversely, if we have an accepting run of $\mathcal{A}_e$ on w, we split the run, and thus the path, into the part before and after taking the new transition added while constructing the automaton. Note that we have to take this transition in order to pass from the initial state, which is in $\mathcal{A}_{e_1}$ part of $\mathcal{A}_e$, to a final state, which is in a $\mathcal{A}_{e_2}$ part of $\mathcal{A}_e$. From this it follows that $w \in L(e)$.

- If $e = e_1^+$, then let again $\mathcal{A}_{e_1}$ be the automaton from the induction hypothesis. Note first that this automaton has at least four states, since $\mathrm{Proj}(e_1) \neq \varepsilon$, where $\mathrm{Proj}(e)$ denotes the projection to the finite alphabet Σ, and transitions going directly from initial to final state can only accept the empty word, so they will not alter computations or acceptance. We let the automaton for e be the same as the one for e_1, but we add the following transitions: for every (d, c, I, w) with $w \in F_1$ and for every (d_1, c', I', w'), where d_1 is the initial state of $\mathcal{A}_{e_1}$, we add $(d, c \wedge c', I \cup I', w')$ to our transition function, thus bypassing the last and

the first state.

Assume now that $(e,w,\sigma) \vdash \sigma'$. Then either $(e_1,w,\sigma) \vdash \sigma'$, so we are done by the induction hypothesis, or $w = w_1 \cdots w_k$ with $k \geq 2$ and valuations $\sigma_1,\ldots,\sigma_{k+1}$ exist such that $(e_1,w_i,\sigma_i) \vdash \sigma_{i+1}$ for $i = 1,\ldots,k$. But then by the induction hypothesis we have computations of $\mathcal{A}_{e_1}$ with σ_i as the initial assignment and σ_{i+1} as final assignment that accept w_i, for $i = 1,\ldots,k$. Note that this actually means that we start with σ, do a computation for w_1, end with σ_2 in the registers, then take the new transition bypassing the end state for this computation and thus starting the computation with σ_2 in the registers(and updating the registers as dictated by the first transition in the new cycle), etc., until we reach σ' after reading w_k, thus accepting w.

For the converse, if $\mathcal{A}_e$ accepts w when started with σ and ended with σ' then we simply split the data path for every time we take the additional transitions added in the construction of $\mathcal{A}_e$. From this we get computations of $\mathcal{A}_{e_1}$ on sub-paths with intermediate valuations. By the induction hypothesis we have acceptance of these subpaths by e_1 with appropriate valuations and thus the membership of the entire path w in $L(e,\sigma,\sigma')$.

This concludes the proof. To see that the construction can be carried out in DLOGSPACE we use the well known fact that DLOGSPACE algorithms can be composed [Papadimitriou, 1993]. □

A natural question to ask is do regular expressions with memory define the same class of queries as register automata. We will prove that this is indeed true when addressing the problem from a language theory point of view in Section 6.2.

Defining queries using Regular expressions with memory

We now deal with the following class of queries.

Definition 4.2.4. *A* regular query with memory *is an expression $Q = x \xrightarrow{e} y$, where e is regular expression with memory.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v,v') such that there is a data path w from v to v' that belongs to $L(e)$.

The class of these queries is denoted by RQM.

Example 4.2.5. *To illustrate some interesting queries expressed by RQMs we again turn to the movie database from Figure 2.3. Same as in the Example 4.1.4 we will assume that each* cast *edge has a corresponding* stars_in *edge going in the other direction.*

- *To express the query from Example 4.1.4 returning actors that have a finite Bacon number we can use $Q = x \xrightarrow{e} y$, where e is given by $(\mathsf{starts\_in} \cdot \mathsf{cast})^+[=\mathsf{Kevin\ Bacon}]$.*

- *To find movies having at least two different actors staring in them we would use the RQM* $Q = x \xrightarrow{e} y$*, where e is* $\downarrow x.\ \mathsf{cast} \downarrow y.\ \mathsf{stars\_in}\ [x^{=}] \cdot \mathsf{cast}[y^{\neq}]$*. Note that here, in addition to the movie we also return one of the actors. The expressions first stores the movie name into the variable x and after that moves to first of the actors. Following this it stores the actor's name into y and moves back to the movie using a* stars_in *edge and checking that it arrived at the same movie by comparing the data value with the one stored into x. Following that the expression simply traverses another* cast *edge, ensuring it reached a different actor by comparing the value in the node to y.*

Using Proposition 4.2.3 combined with Theorem 4.1.6 we immediately obtain:

Corollary 4.2.6. *Data complexity of RQM queries is in* NL.

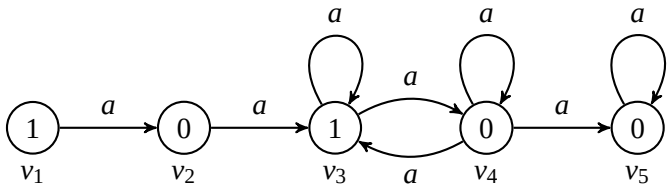
From the same connection we also get the upper bound (PSPACE) for combined complexity. It turns out that we can achieve PSPACE-hardness with expressions with memory. Thus, we have

Theorem 4.2.7. *Combined complexity of evaluating RQM queries is* PSPACE*-complete.*

Proof. The PSPACE upper bound follows from Theorem 4.1.6 and Proposition 4.2.3. Thus we only have to prove PSPACE-hardness. For this we do a reduction from regular automata nonuniversality problem. The idea is to simulate on the fly reachability testing in the powerset automaton by using two sets of variables, each of the size of the automaton, for coding the current and the next state.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$ be a finite state automaton, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$. We will construct a fixed graph G with 5 nodes, containing two distinguished nodes s and t in G and construct, in polynomial time, a regular expression with memory e,of length $O(n \times |\Sigma|)$, such that $(s,t) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$, where $Q = x \xrightarrow{e} y$.

The graph G is shown below:



We now set $s = v_1$ and $t = v_5$.

Since we are trying to demonstrate nonuniversality of the automaton $\mathcal{A}$ we simulate reachability checking in the powerset automaton for $\overline{\mathcal{A}}$. To do so we designate two distinct data values, t and f, and code each state of the powerset automaton as an n-bit sequence of t/f values, where the ith bit of the sequence is set to t if the state q_i is included in our state of $\overline{\mathcal{A}}$. Since

we are checking reachability we will need only to remember the current and the next state of $\overline{\mathcal{A}}$. In what follows we will code those two states using variables $s_1,\ldots,s_n$ and $w_1,\ldots,w_n$ and refer to them as stable tape and work tape. Our expression e will code data paths that describe successful runs of $\overline{\mathcal{A}}$ by demonstrating how one can move from one state of this automaton to another (as witnessed by their codes in stable and work tapes), starting with the initial and ending in a final state.

We will define several expressions and explain their role. We will use two sets of variables, s_1 through s_n and $w_1,\ldots,w_n$ to denote stable and work tape (i.e. current and next state in the powerset automaton). All of these variables will only contain two values, t and f, which are bound in the beginning and that will correspond to 0 and 1 in the graph G.

The first expression we need is:

$$\texttt{init} := \downarrow t.a[t^{\neq}]\downarrow f.a[t^{=}]\downarrow s_1.a[f^{=}]\downarrow s_2\ldots a[f^{=}]\downarrow s_n.a.$$

This expression codes two different values as t and f and initializes stable tape to contain encoding of initial state (the one where only initial state from $\mathcal{A}$ can be reached). That is, a data path is in the language of this expression if and only if it starts with two different data values and continues with n data values that form a sequence in 10^*.

$$\texttt{end} := a[f^{=} \wedge s_{i_1}^{=}]\cdot a[f^{=} \wedge s_{i_2}^{=}]\cdots a[f^{=} \wedge s_{i_k}^{=}], \text{ where } F = \{q_{i_1},\ldots,q_{i_k}\}.$$

This expression is used to check that we have reached a state not containing any final state from the original automaton. That is, a data path is in $L(\texttt{end})$ if and only if it consists of k data values, all equal to f and where value stored in s_{i_j} also equals f, for $j = 1\ldots k$.

Next we define expressions that will reflect updating of the work tape according to the transition function of $\mathcal{A}$. Assume that $\delta(q_i,b) = \{q_{j_1},\ldots,q_{j_l}\}$. We define

$$u_{\delta(q_i,b)} := \left(a[t^{=} \wedge s_i^{=}]\cdot a[t^{=}]\downarrow w_{j_1}\ldots a[t^{=}]\downarrow w_{j_l}.a\right) + a[f^{=} \wedge s_i^{=}].$$

Also, if $\delta(q_i,b) = \emptyset$ we simply put $u_{\delta(q_i,b)} := \varepsilon$.

This expression will be used to update the work tape by writing true to corresponding variables if the state q_i is tagged with t on the work tape (and thus contained in the current state of $\overline{\mathcal{A}}$). If it is false we skip the update.

Since we have to define update according to all transitions from all the states corresponding to chosen letter we get:

$$\texttt{update} := \bigvee_{b\in\Sigma} \bigwedge_{q_i\in Q} u_{\delta(q_i,b)}.$$

This simply states that we non deterministically pick the next symbol of the word we are guessing and move to the next state accordingly.

We still have to ensure that the tapes are copied at the beginning and end of each step, so we define:

$$\texttt{step} := (a[f^{=}]{\downarrow}w_1 \ldots a[f^{=}]{\downarrow}w_n.a) \cdot \texttt{update} \cdot (a[w_1^{=}]{\downarrow}s_1 \ldots a[w_n^{=}]{\downarrow}s_n.a).$$

This simply initializes the work tape at the beginning of each step, proceeds with the update and copies the new state to stable tape. Note the few odd a's at the end of the expressions. These will not affect what we what to achieve and are here for syntactical reasons(to get a proper expression).

Finally we have

$$e := \texttt{init} \cdot (\texttt{step})^* \cdot \texttt{end}.$$

Here we use $\texttt{step}^*$ as abbreviation for $\texttt{step}^+ + \varepsilon$.

We claim that for $Q = x \xrightarrow{e} y$, we have $(s,t) \in Q(G)$ if and only if $L(\mathcal{A}) \neq \Sigma^*$.

Assume first that $L(\mathcal{A}) \neq \Sigma^*$. This means that there is a path from the initial to the final state in the powerset automaton for $\overline{\mathcal{A}}$. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This path can in turn be described by pairs of assignment of values t/f to stable and work tape, where each transition is witnessed by the corresponding letter of the alphabet. But then the path from s to t in G that belongs to $L(e)$ is the one that first initializes the stable tape (i.e. the variables $s_1, \ldots, s_n$) to initial state of the powerset automaton, then runs the updates of the tape according to w and finally ends in a state where all variables corresponding to end states of $\mathcal{A}$ are tagged f. Note that we can describe this path in G, since we start in s and put 1 into t in node v_1, 0 into f in node v_2. After that 1 is assigned to s_1 in v_3 and 0 to $s_2, \ldots, s_n$ by looping through v_4. After that each transition is reflected by going through v_3 and v_4 as necessary, to update tapes with t/f and finally going to v_5 and looping there to check that all s_i's corresponding to end states are tagged f.

Conversely, each path from s to t in $L(e)$ corresponds to a run of the powerset automaton for $\overline{\mathcal{A}}$. That is, the part of path corresponding to `init` sets the initial state. Then the part of this path that corresponds to $\texttt{step}^*$ corresponds to updating our tapes in a way that properly codes one step of powerset automaton. Finally, `end` denotes that we have reached a state where all end states of $\mathcal{A}$ have been tagged by f, thus, an accepting state for $\overline{\mathcal{A}}$. □

The question is whether we can reduce this complexity – ideally to PTIME, but at least to NP, to match the combined complexity of conjunctive queries. The following corollary (to the proof of Theorem 4.2.7) shows that many restrictions will not work.

Corollary 4.2.8. *Combined complexity of evaluating RQM queries remains* PSPACE*-hard for expressions that use at most one* $^+$ *and* $\neq$ *symbol, are specified over a singleton alphabet* $\Sigma = \{a\}$*, and are evaluated over a fixed graph.*

In one case, we can lower the complexity.

Proposition 4.2.9. *Combined complexity of RQM queries whose regular expressions do not have subexpressions of the form e^+ is* NP*-complete.*

Proof. Recall that for $e \in \mathrm{REG}(\Sigma[x_1,\ldots,x_k])$, by $\mathrm{Proj}(e)$ we denote the projection of e to the finite alphabet Σ.

First we show NP-membership. Since we do not use $^+$ we know that every data path in the language of expression e uses at most $|\mathrm{Proj}(e)|$ letters and one more data value. Assume now that we are given a data graph G, two nodes $s,t \in G$ and an expression with memory e. To see if $(s,t) \in Q(G)$, for $Q = x \xrightarrow{e} y$, we use the following algorithm. First compute the register automaton $\mathcal{A}_e$ for e. Note that this can be done in DLOGSPACE. Then nondeterministically guess a data path w_π in G from s to t that is of length at most $|\mathrm{Proj}(e)|$. Now also guess $2|\lambda(w_\pi)|+1$ states of $\mathcal{A}_e$ and check that the path w_π is accepted by $\mathcal{A}_e$, as witnessed by this sequence of states, and thus is in $L(e)$. It is straightforward to see that this can be done in polynomial time and since our guesses are of polynomial (in fact linear) size we get the desired result.

For hardness we do a reduction from k-CLIQUE. This problem asks for a given graph G and a number k, to determine if G has a clique of size at least k.

Suppose we are given an undirected graph G and a number k. We will construct a data graph G' with $|G|+2$ nodes, select two nodes $s,t \in G'$ and construct a regular expression with memory e_k of size $O(k^2)$ such that G contains a k-clique if and only if there is a data path from s to t in G' that satisfies e_k.

Take $\Sigma = \{a,b\}$ and make G directed by adding edges in both directions for every edge in G. Label all the edges by a and add two more nodes s and t. Add an edge from s to every other node except s,t and label them with b. Also add an edge from every node in G to t and label them by b. To finish the construction just add a different data value to every node. We call the resulting graph G'.

To define e_k we use an auxiliary expression δ_i defined as:

$$\delta_i := a[x_1^=] \cdot a[x_i^=] \cdot a[x_2^=] \cdot a[x_i^=] \ldots a[x_{i-1}^=] \cdot a[x_i^=].$$

This expression will simply allow us to test that the current node is connected to all nodes previously selected in our potential clique.
Now we can define e_k inductively as follows:

- $e_1 := b \cdot \downarrow x_1.a[x_1^{\neq}]$,
- $e_2 := e_1 \cdot \downarrow x_2.a[x_1^{\neq} \wedge x_2^{\neq}]$,
- $e_i := e_{i-1} \cdot \downarrow x_i.\delta_i \cdot a[x_1^{\neq} \wedge \ldots \wedge x_i^{\neq}]$, for $i = 3,\ldots,k-1$ and
- $e_k := e_{k-1} \cdot \downarrow x_k.\delta_k \cdot b$.

Next we show that there is a k-clique in G iff there is a data path form s to t in G' that satisfies e_k.

Suppose first that there is a k-clique in G. Then we simply move from s to an arbitrary point in that clique using the b labeled edge and traverse the clique back and forth until we reach the k-th element of the clique. Note that starting from the third element, whenever we select a different node in the clique we have to move back and forth between this node and all previously selected ones to satisfy δ_i, but since we have a clique this is possible. Finally, after selecting the last node and verifying that it is connected to all the others we move to t using a b labelled edge.

Now suppose that there is a data path from s to t in G' that satisfies e_k. This means that we will be able to select k different nodes $n_1, \ldots, n_k$ in G with data values stored in $x_1, \ldots, x_k$. Since all data values in the graph are different they also act as ids. Now take any two n_i, n_j with $i < j \leq k$. Then we know that n_i and n_j are connected in G because after selecting n_j we have to go through δ_j which contains $a[x_i^{=}] \cdot a[x_j^{=}]$ and since no two data values in G are the same this means that we have an edge between n_i and n_j. This completes the proof. □

The restriction, while achieving better combined complexity, is too strong, as it effectively restricts one to languages of data paths whose projections on Σ^* are finite. All the examples we saw earlier use subexpressions e^+. So if we want to achieve tractability, we need to look at a very different way of restricting expressions. This is what we do in the next two sections.

4.3 Regular queries with binding (RQBs)

When examining Regular expressions with memory one asymmetry becomes apparent quite quickly. Namely, they do not define the scope of variables. To illustrate this, consider the following regular expression with memory:

$$\downarrow x.a \cdot (a[x^{\neq}] \downarrow x.a)^* \cdot a[x^{=}].$$

This expression re-binds variable x inside the scope of another binding, and then crucially, when this happens, the original binding of x is *lost*! Such expressions really mimic the behaviour of register automata, which makes them more procedural than declarative. Although this behaviour is necessary to show equivalence between register automata and regular expressions with memory, as we will demonstrate in Section 6.2, it goes against the usual practice of writing logical expressions and programs that have bound variables.

Therefore it makes sense to study expressions that have proper scoping rules defined. In this section we show that using such expressions makes writing graph queries more natural, however, it does not garner any decrease in computational requirements when querying graphs.

In a later section we will also study these expressions from a language theory point of view and show that they are strictly weaker than register automata.

Definition 4.3.1 (Expressions with binding)**.** *Let Σ be a finite alphabet and $x_1,\ldots,x_k$ a set of variables. Then* regular expressions with binding *are defined by the grammar:*

$$e \;:=\; \varepsilon \mid a \mid e+e \mid e\cdot e \mid e^{+} \mid e[c] \mid \downarrow\overline{x}.\{e\} \tag{4.2}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1,\ldots,x_k$.

As before we will assume that all the expressions are well-formed.

The class of well-formed regular expressions with binding is denoted by $\mathrm{REB}(\Sigma[x_1,\ldots,x_k])$.

Note that the scope of variables $\overline{x}$ in an expression of the form $\downarrow\overline{x}.\{e\}$ is explicitly denoted by parenthesis which make it extend only in the subexpression e. For example the last occurrence of x in $\downarrow x.\{(a[x^{\neq}])^*\}\cdot a[x^{=}]$ is outside of the scope of $\downarrow x$ and will thus not be compared to the first data value in the word, as would be the case in a regular expression with memory.

Since regular expressions with binding have proper scoping, they also have the usual notion of free and bound variables. A variable x is bound if it occurs in the scope of some $\downarrow x$ operator and free otherwise. More precisely, free variables of an expression are defined inductively: ε and a have no free variables, in $e[c]$ all variables occurring in c are free, in e_1+e_2 and $e_1\cdot e_2$ the free variables are those of e_1 and e_2, the free variables of e^{+} are those of e, and the free variables of $\downarrow\overline{x}.\{e\}$ are those of e except $\overline{x}$. We will write $e(x_1,\ldots,x_l)$ if $x_1,\ldots,x_l$ are the free variables in e.

A valuation on the variables $x_1,\ldots,x_k$ is a partial function $\nu : \{x_1,\ldots,x_k\} \mapsto \mathcal{D}$. For a valuation ν, we write $\nu[x_i \leftarrow d]$ to denote the valuation ν' obtained by fixing $\nu'(x_i) = d$ and $\nu'(x) = \nu(x)$ for all other $x \neq x_i$. Likewise, we write $\nu[\overline{x} \leftarrow \overline{d}]$ for a simultaneous substitution of values from $\overline{d} = (d_1,\ldots,d_l)$ for variables $\overline{x} = (x_1,\ldots,x_l)$ and $\nu[\overline{x} \leftarrow d]$ when $d_1 = \ldots = d_l = d$. Also notation $\nu(\overline{x}) = \overline{d}$ means that $\nu(x_i) = d_i$ for all $i \leq l$.

Let $e(\overline{x})$ be from $\mathrm{REB}(\Sigma[x_1,\ldots,x_k])$. A valuation ν is compatible with e, if $\nu(\overline{x})$ is defined.

The semantics of a regular expression with binding e is given with respect to a compatible valuation $\nu : \{x_1,\ldots,x_k\} \mapsto \mathcal{D}$ and it denotes the set of data paths $L(e,\nu)$ inductively as follows:

- $L(\varepsilon,\nu) = \{d \mid d \in \mathcal{D}\}$.
- $L(a,\nu) = \{dad' \mid d,d' \in \mathcal{D}\}$.
- $L(e+e',\nu) = L(e,\nu) \cup L(e',\nu)$.
- $L(e\cdot e',\nu) = L(e,\nu)\cdot L(e',\nu)$.
- $L(e^{+},\nu) = L(e,\nu)^{+}$.
- $L(e[c],\nu) = \{d_1a_1\ldots a_kd_{k+1} \in L(e,\nu) | d_{k+1},\nu \models c\}$.

- $L(\downarrow \overline{x}.\{e\}, \nu) = \{d_1 a_1 \dots a_k d_{k+1} | d_1 a_1 \dots a_k d_{k+1} \in L(e, \nu[\overline{x} \leftarrow d_1])\}$.

If an expression has no free variables it is called *closed*. When dealing with closed regular expressions with binding it is not necessary to specify a valuation. We can thus talk about $L(e)$, the language of data paths defined by a closed expression e.

Next we give a few examples of regular expressions with binding and languages they define.

Example 4.3.2. *Here we give some examples of data path languages definable by regular expressions with binding. These will be similar to the ones given in Example 4.2.2 to demonstrate that one can define some interesting properties of data paths even with the restrictions that proper scoping rules impose.*

1. *The language where all data values differ from the first one is given by the expression* $\downarrow x.\{(a[x^{\neq}])^+\}$.
2. *The language where first data value differs from the last one is given by* $\downarrow x.\{a^* a[x^{\neq}]\}$.
3. *The language where two data values are equal is given by* $a^* \cdot \downarrow x.\{a^* a[x^{=}]\} \cdot a^*$.
4. *The language where each data value equals the first one, but the second value is different from 5 is given by* $\downarrow x.\{a[5^{\neq} \wedge x^{=}](a[x^{=}])^*\}$.

It is straightforward to see that regular expressions with binding are subsumed by register automata and expressions with memory. Moreover, going from expressions with binding to expressions with memory (and thus register automata) is trivially achieved by renaming of variables. For instance regular expression with binding $\downarrow x.\{a[x^{=}] \cdot \downarrow x.\{a[x^{\neq}]\} \cdot a[x^{=}]\}$ is equivalent to regular expression with memory $\downarrow x.a[x^{=}] \cdot \downarrow y.a[y^{\neq}] \cdot a[x^{=}]$. We thus obtain the following.

Proposition 4.3.3. *For every regular expression with binding e we can construct an equivalent regular expression with memory* e' *in* DLOGSPACE.

When comparing the formalisms in Section 6.3 we will show that the converse is not true.

Queries based on expressions with binding

Similarly as when dealing with RQMs, we now define a class of queries based on regular expressions with binding.

Definition 4.3.4. *A* regular query with binding *is an expression* $Q = x \xrightarrow{e} y$*, where e is a closed regular expression with binding.*

Given a data graph G, the result of the query $Q(G)$ *consists of pairs of nodes* (v, v') *such that there is a data path w from v to* v' *that belongs to* $L(e)$.

The class of these queries is denoted by RQB.

Example 4.3.5. *To give an example of an RQB query we note that both expressions in Example 4.2.5 can be expressed with a properly defined scope. While the expression finding people with a finite Bacon number already is a regular expression with binding, to find movies having at least two different actors we use* $Q = x \xrightarrow{e} y$*, where e equals* $\downarrow x.\{\mathsf{cast}\downarrow y.\{\mathsf{stars\_in}[x^{=}] \cdot \mathsf{cast}[y^{\neq}]\}\}$.

Using Proposition 4.3.3 combined with Corollary 4.2.6 we immediately obtain:

Corollary 4.3.6. *Data complexity of RQB queries is* NL*-complete.*

Seeing how scoping puts a restriction on the expressive power of languages, and since the expressions used to show hardness in Theorem 4.2.7 use the fact that no scope is defined for the variables they use, one might hope that query evaluation for *RQB*s might be more efficient that for *RQM*s. However, we show next that this is not the case.

Theorem 4.3.7. *The combined complexity of query evaluation for RQBs is* PSPACE*-complete.*

Proof. Note that the upper bound follows from Proposition 4.3.3 and Corollary 4.2.6.

Now we prove the PSPACE-hardness of our theorem. The reduction is form QBF.

Let

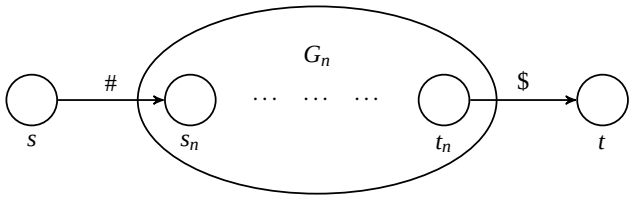
$$\begin{aligned} \Psi &= \forall x_n \exists y_n \ldots \forall x_1 \exists y_1 \, \varphi \\ \varphi &= (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge (\ell_{2,1} \vee \ell_{2,2} \vee \ell_{2,3}) \wedge \cdots \wedge (\ell_{m,1} \vee \ell_{m,2} \vee \ell_{m,3}) \end{aligned}$$

where each $\ell_{i,j}$ is a literal. We call a literal $\ell_{i,j}$ a *negative* literal, if it is a negation of a variable. Otherwise, we call it a *positive* literal.

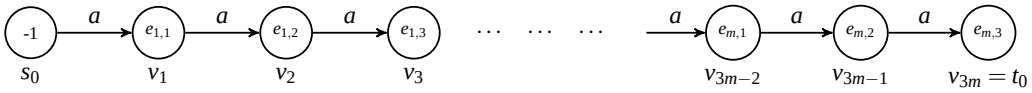
For each $i \in \{0,1,\ldots,n\}$, we will denote $\Psi_i = \forall x_i \exists y_i \ldots \forall x_1 \exists y_1 \varphi$. Hence, $\Psi_0 = \varphi$ and $\Psi_n = \Psi$. We are going to construct (in polynomial time) a graph G, two nodes $s,t \in G$ and a closed regular expression with binding r such that for the *RQB* $Q = x \xrightarrow{r} y$ it holds that

$$\Psi \text{ is true} \quad \text{if and only if} \quad (s,t) \in Q(G).$$

Construction of the graph G and the two nodes $s,t \in G$*:* The graph G is a data graph over $\Sigma = \{a,b,\#,\$\}$. Its construction is done inductively on $i \in \{0,1,\ldots,n\}$, where G_i, s_i, t_i are constructed from Ψ_i. The desired graph G and the two nodes $s,t \in V(G)$ is the following graph.



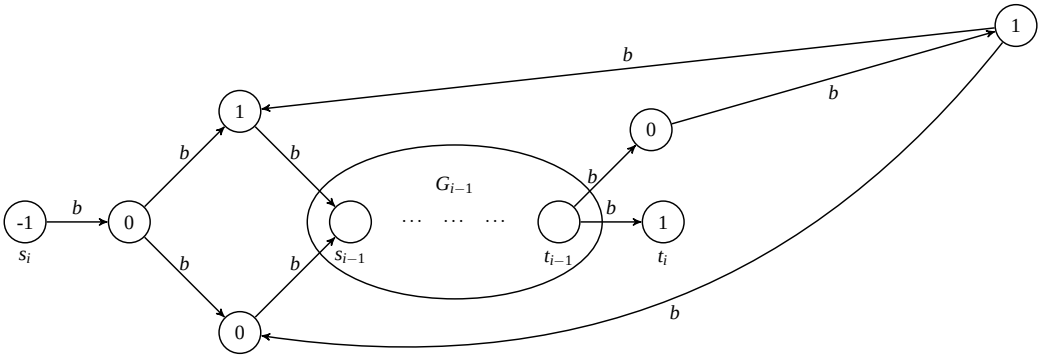
The construction of G_i, s_i, t_i is inductive on i. The graph G_0 and the two vertices s_0, t_0 are as follows.



where

$$e_{i,j} = \begin{cases} 1 & \text{if the literal } \ell_{i,j} \text{ is positive} \\ 0 & \text{if the literal } \ell_{i,j} \text{ is negative} \end{cases}$$

Now we show the construction of G_i, s_i, t_i . Suppose we already constructed $G_{i-1}, s_{i-1}, t_{i-1}$. Then G_i, s_i, t_i is as follows.



The construction of the expression r: In the following we are going to show the construction of the expression r. We first show how to construct an auxiliary expression r_i, for each $i = 0, 1, \ldots, n$, which is based on the formula Ψ_i. The desired expression r is then defined as $r = \# \cdot r_n \cdot \$$.

The expression r_i is defined inductively on $i = 0 \ldots n$. First we set

$$r_0 = \text{clause}_1 \cdot \text{clause}_2 \cdots \text{clause}_m,$$

where each clause_i is defined as follows.

$$\text{clause}_i = a[x_{i,1}^{=}] \cdot a \cdot a + a \cdot a[x_{i,2}^{=}] \cdot a + a \cdot a \cdot a[x_{i,3}^{=}]$$

and $x_{i,1}, x_{i,2}, x_{i,3}$ are the variables in the literals $\ell_{i,1}, \ell_{i,2}, \ell_{i,3}$, respectively.

Now, assuming we have the expression r_{i-1}, we define r_i as follows.

$$r_i = \left(b{\downarrow}x_i.\{b{\downarrow}y_i.\{b \cdot r_{i-1}\} \cdot b[x_i^{=}]\} \right)^*.$$

Finally we set $r = \# \cdot r_n \cdot \$$.

It is straightforward to verify that the construction of both the data graph G and the expression r runs in time polynomial in the length of the formula Ψ.

Remark 4. *For every $i = 0,1,\ldots,n$,*

- *the formula Ψ_i has free variables $x_{i+1},y_{i+1},\ldots,x_n,y_n$;*
- *the expression r_i has free variables $x_{i+1},y_{i+1},\ldots,x_n,y_n$.*

Moreover, for a tuple $\overline{d} \in \{0,1\}^{2(n-i)}$, we write $\Psi_i(\overline{d})$ to denote the formula Ψ_i in which the variables $x_{i+1},y_{i+1},\ldots,x_n,y_n$ are assigned with $\overline{d}$. We also define the query $Q_i^{\overline{d}} = x \xrightarrow{r_i,\nu(\overline{d})} y$, where $\nu(\overline{d})$ is the valuation assigning values in $\overline{d}$ to $x_{i+1},y_{i+1},\ldots,x_n,y_n$. Then we have $(v,v') \in Q_i^{\overline{d}}(G)$ if and only if there is a path π from v to v' in G such that $w_\pi \in L(r_i,\nu(\overline{d}))$. Note that the query here is dependant on the valuation ν.

To prove that Ψ is true if and only if $(s,t) \in Q(G)$, we prove the following claim.

Claim 4.3.8. *For each $i = 0,1,\ldots,n$ and for every tuple $\overline{d} \in \{0,1\}^{2(n-i)}$, $\Psi_i(\overline{d})$ is true if and only if $(s_i,t_i) \in Q_i^{\overline{d}}(G_i)$.*

Proof. The proof is by induction on i. The basis is $i = 0$. We have to prove that $\Psi_i(\overline{d})$ is true if and only if $(s_0,t_0) \in Q_0^{\overline{d}}(G_0)$.

Let for each $k = 1,\ldots,m$ and $j = 1,2,3$, we write $d_{k,j}$ to denote the 0-1 value assigned to the variable in the literal $\ell_{k,j}$. Let ν denote the valuation where $\nu(x_1),\nu(y_1),\ldots,\nu(x_n),\nu(y_n)$ are assigned with $\overline{d}$, respectively. Then, we have

$$\Psi_0(\overline{d}) \text{ is true}$$
$$\Updownarrow$$
$$\text{every clause } (\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3}) \text{ is true under the assignment } \nu$$
$$\Updownarrow$$
$$\text{for each } k = 1,\ldots,m, \text{ there exists } j \in \{1,2,3\} \text{ such that}$$
$$d_{k,j} = \begin{cases} 1 & \text{if } \ell_{k,j} \text{ is positive} \\ 0 & \text{if } \ell_{k,j} \text{ is negative} \end{cases}$$
$$\Updownarrow$$
$$\text{for each } k = 1,\ldots,m, w_{\pi_k} \in L(\text{clause}_i,\nu) \text{ where}$$
$$\pi_k = v_{3k+0}av_{3k+1}av_{3k+2}av_{3k+3}$$
$$\Updownarrow$$
$$(s_0,t_0) \in Q_0^{\overline{d}}(G_0)$$

For the induction hypothesis, we assume that $\Psi_i(\overline{d})$ is true if and only if $(s_i,t_i) \in Q_i^{\overline{d}}(G_i)$.

For the induction step, we prove the claim for $i+1$, which follows from the following equality.

$$\Psi_{i+1}(\overline{d}) \text{ is true}$$
$$\Updownarrow$$
$$\text{there exist } e_0, e_1 \in \{0,1\} \text{ such that } \Psi_i(\overline{d}0e_0) \text{ and } \Psi_i(\overline{d}1e_1) \text{ are true}$$
$$\Updownarrow$$
$$\text{there exist } e_0, e_1 \in \{0,1\} \text{ such that } (s_i,t_i) \in Q_i^{\overline{d}0e_0}(G_i) \text{ and } (s_i,t_i) \in Q_i^{\overline{d}1e_1}(G_i) \in Q(r_i,G_i).$$
$$\Updownarrow$$
$$\text{there exists a path } \pi \text{ from } s_{i+1} \text{ to } t_{i+1} \text{ such that } w_\pi \in L(r_{i+1}, \nu(\overline{d}))$$

The last inequality follows from the definition of r_{i+1}, where

$$r_{i+1} = \Big(b{\downarrow}x_{i+1}.\{b{\downarrow}y_{i+1}.\{b\cdot r_i\}\cdot b[x_{i+1}^{=}]\}\Big)^*.$$

and to go from the vertex s_{i+1} to t_{i+1}, the path π has to go thorough G_i at least twice: once when the variable x_{i+1} is assigned with 0 and at least once when the variable x_{i+1} is assigned with 1. Thus, we have $\Psi_{i+1}(\overline{d})$ is true if and only if $(s_{i+1},t_{i+1}) \in Q_{i+1}^{\overline{d}}(G_{i+1})$. □

This concludes the proof of the hardness part, hence, our theorem.

4.4 Regular queries with data tests (RQDs)

The class of regular expressions for data paths that lets us lower the combined complexity of queries to PTIME permits testing for equality or inequality of data values at the beginning or the end of a data (sub)path. For example, $(\Sigma^+)_{\neq}$ denotes the set of all data paths having different first and last data values. The language L_{eq} of data paths on which two data values are the same is given by $\Sigma^* \cdot (\Sigma^+)_{=} \cdot \Sigma^*$: it checks for the existence of a nonempty subpath (with label in Σ^+) such that the nodes at the beginning and at the end of this subpath have the same data value, indicated by subscript $=$.

To allow for constants we will use *simplified conditions*. These are simply conjunctions of the form $e^{=}$ and $e^{\neq}$, where e ranges over $\mathcal{D}$. Then a data value d satisfies a simplified condition c, denoted $d \models c$, if $\tau, d \models c$, where τ is an empty assignment. Note that the valuation itself is irrelevant here.

Definition 4.4.1 (Expressions with equality)**.** *Let Σ be a finite alphabet. Then* regular expressions with equality *are defined by the grammar:*

$$e \;:=\; \varepsilon \mid a \mid e+e \mid e\cdot e \mid e^+ \mid e[c] \mid e_{=} \mid e_{\neq} \tag{4.3}$$

where a ranges over alphabet letters and c is a simplified condition.

The language $L(e)$ of data paths denoted by a regular expression with equality e is defined as follows.

- $L(\varepsilon) = \{d \mid d \in \mathcal{D}\}$.
- $L(a) = \{dad' \mid d,d' \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e[c]) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_n \models c\}$.
- $L(e_=) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 = d_n\}$.
- $L(e_{\neq}) = \{d_1 a_1 \ldots a_{n-1} d_n \in L(e) \mid d_1 \neq d_n\}$.

These expressions sacrifice the ability to store data values, making it only possible to check for (in)equality at the start and the end of chosen subexpressions. The only exception is testing against constants, but since these tests are so natural from a database point of view we include them in the definition. Looking at Example 4.2.2, all languages except the first can be defined by regular expressions with memory. We already saw how to do the language L_{eq}; the expression $\downarrow x.(ab)^+[x^{\neq}]$ is equivalent to $(ab)^+_{\neq}$. The expression $\downarrow x.(a[x^{\neq}])^+$ describing the language of data paths in which all data values are different from the first one, requires checking a condition multiple times. We now show that this goes beyond the power of expressions with equality, which are strictly weaker than expressions with memory.

Proposition 4.4.2. *1. For each regular expression with equality, there is an equivalent regular expression with memory.*

2. For the regular expression with memory $\downarrow x.(a[x^{\neq}])^+$ there is no equivalent regular expression with equality.

Proof. For first item it is enough to observe that for expressions of the kind $e_=$ and $e_{\neq}$, where e is an ordinary regular expression, the expressions with memory $\downarrow x.e[x^{=}]$ and $\downarrow x.e[x^{\neq}]$ denote the same language of data paths. From this it is straightforward to construct a translation of arbitrary regular expression with equality e to regular expression with memory by doing the above mentioned construction bottom-up, starting from subexpressions of e and using a new variable for each subexpression of the form $e'_=$ or $e'_{\neq}$.

To prove the second claim we introduce a new kind of automata, called weak register automata, show that they capture regular expressions with equality and that they can not express the language $\downarrow x.(a[x^{\neq}])^+$ of a-labeled data paths on which all data values are different from the first one.

The main idea behind weak register automata is that they erase the data value that was stored in the register once they make a comparison, thus rendering the register empty. We denote this by putting a special symbol $\perp$ from $\mathcal{D}$ in the register. Since they have a finite number of registers, they can keep track of only finitely many positions in the future, so in the case of our language, they can only check that a fixed finite number of data values is different from the first one. We proceed with formal definitions.

The definition of weak k-register data path automaton is the same as in the Definition 6.1.1. The only explicit change we make is that we now assume that C_k contains a special symbol ε, that will allow us to simply skip the data value, without doing any comparisons (previously we have been using a simple tautology such as $x_1^{=} \vee x_1^{\neq}$, or an additional register to emulate this). Thus we simply add $\tau, d \models \varepsilon$, for every valuation τ and data value d, to semantics of C_k. We will also assume that the initial configuration is always empty.

Definition of configuration remains the same as before, but the way we move from one configuration to another changes.

From a configuration $c = (j, q, \tau)$ we can move to a configuration $c' = (j+1, q', \tau')$ if one of the following holds:

- the jth symbol is a letter a, and there is a transition $(q, a, q') \in \delta_w$; or
- the current symbol is a data value d, and there is a transition $(q, c, I, q') \in \delta_d$ such that $d, \tau \models c$ and τ' coincides with τ except that every register mentioned in c is set to be empty (i.e. to contain $\perp$) and the ith component of τ' is set to d whenever $i \in I$.

The second item simply tells us that if we used a condition like $c = x_3^{=} \wedge x_7^{\neq}$ in our transition, we would afterwards erase data values that were stored in registers 3 and 7. Note that we can immediately rewrite these registers with the current data value.

The notion of acceptance and an accepting run is the same as before.

We now show that weak register automata can not recognize the language L of all data paths where first data value is different from all other data values, i.e. the language denoted by the expression $\downarrow x.(a[x^{\neq}])^{+}$.

Assume to the contrary, that there is some weak $k-$register data path automaton $\mathcal{A}$ recognizing L. Since data path $w_\pi = d_1 a d_2 a \dots d_k a d_{k+1} a d_{k+2}$, where d_is are pairwise different and do not appear in any condition in $\mathcal{A}$, is in L, there is an accepting run of $\mathcal{A}$ on w_π. The idea behind the proof is that $\mathcal{A}$ can check that only the first $k+1$ positions have different data value from the first.

First we note a few things. Since every data value in the path w_π is different, no $=$ comparisons can be used in conditions appearing in this run (otherwise the condition test would fail and the automaton would not accept). This also must hold for constants appearing in the conditions, since no d_is appear in them.

Now note that since we have only k registers, and with every comparison we empty the corresponding registers one of the following must occur:

- There is a data value $1 < i < k+2$ such that the condition used when processing this data value is ε. In this case we simply replace d_i by d_1 and get an accepting run on a word that has the first data value repeated – a contradiction. Note that we could store d_i in that transition, but since afterwards we only test for inequality this will not alter the rest of the computation.
- There is a data value such that when the automaton reads it it does not use any register with the first data value, i.e. d_1, stored. Note that this must happen, because at best we can store the first data value in all the registers at the beginning of our run, but after that each time we read a data value and compare it to the first we lose the first data value in this register. But then again we can simply replace this data value with d_1 and get an accepting run (just as before, if this data value gets stored in this transition and then used later it can only be used in a $\neq$ comparison, which is also true for d_1, so the run remains accepting). Again we arrive at a contradiction.

This shows that no weak register automaton can recognize the language L.

To complete the proof of Proposition 4.4.2 we still have to show the following:

Lemma 4.4.3. *For every regular expression with equality e there exists a weak k-register automaton $\mathcal{A}_e$, recognizing the same language of data paths, where k is the number of times $=, \neq$ symbols appear in e.*

The proof of the lemma is almost identical to the proof of Proposition 4.2.3. We can view this as introducing a new variable for every $=, \neq$ comparison in e and act as the subexpression $e'_{=}$ reads $\downarrow x.e'[x^{=}]$ and analogously for $\neq$. Note that in this case all variables come with their scope, so we do not have to worry about transferring register configurations from one side of the construction to another (for example when we do concatenation). The underlying automata remain the same. □

Queries based on Regular expressions with equality

We now deal with the following queries.

Definition 4.4.4. *A* regular query with data tests *is an expression $Q = x \xrightarrow{e} y$, where e is a regular expression with equality.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(e)$.

The class of these queries is denoted by RQD.

Example 4.4.5. *Coming back to the database from Figure 2.3, we can now ask the following queries.*

- *The query asking for people with a finite Bacon number is again the same as in Example 4.2.5.*
- *Query that checks if there is a movie in the database with at least two different actors is defined by* $Q = x \xrightarrow{e} y$*, with* $e := (\mathsf{stars\_in} \cdot \mathsf{cast})_{\neq}$*. Note that a nonempty answer to this query merely signifies that such a movie exists. To actually retrieve the movie we would need to use conjunctive queries with RQDs as atoms (Section 5.2).*

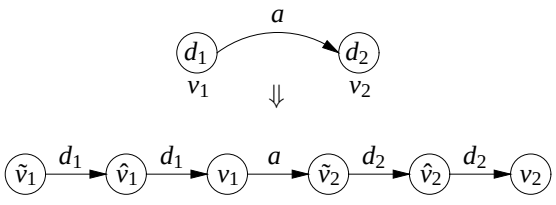
Combining Propositions 4.2.3 and 4.4.2 we see that the power of regular expressions with equality is subsumed by register automata; hence combined with Theorem 4.1.6 we immediately obtain:

Corollary 4.4.6. *Data complexity of RQD queries is in* NL.

We now show that combined complexity for *RQD* queries is tractable, i.e., is even better than the combined complexity of conjunctive queries. Our outline of the polynomial-time algorithm is as follows. We start with a data graph $G = \langle V, E, \rho \rangle$ whose data values form a (finite) set $D \subset \mathcal{D}$ and a regular expression with equality e.

1. We first show that we can efficiently generate a context-free grammar $\mathcal{G}_{e,D}$ whose language corresponds to the set of all data paths from $L(e)$ whose data values are in D. More precisely, every word in $L(\mathcal{G}_{e,D})$ will be of the form $d_1 a_1 d_2 d_2 a_2 d_3 d_3 \ldots d_{n-1} d_{n-1} a_{n-1} d_n$, where $d_i \in D$ and $a_i \in \Sigma$. We say that this word, in which each data value, except the first and the last, appears twice, corresponds to the data path $d_1 a_1 d_2 a_2 d_3 \ldots a_{n-1} d_n$.
2. We then convert $\mathcal{G}_{e,D}$, in polynomial time, into an equivalent PDA $\mathcal{A}(\mathcal{G}_{e,D})$.
3. Given two nodes v, v' in G, we construct an NFA $\mathcal{A}_{G,v,v'}$. To do so we first define a graph $G' = \langle V', E' \rangle$ that will reflect the fact that all data values from G have to be doubled if they appear on a path as intermediate nodes. We define $G' = \langle V', E' \rangle$ as follows:
   - $V' = V \cup \{\tilde{u}, \hat{u} \mid u \in V\} \cup \{s, t\}$
   - $E' = \{(v_1, a, \tilde{v}_2) \mid (v_1, a, v_2) \in E\}$
     $\bigcup \{(\tilde{u}, \rho(u), \hat{u}), (\hat{u}, \rho(u), u) \mid u \in V\}$

   Similarly as when dealing with register automata we triple each node and add an edge between new nodes that will reflect the fact that every intermediate data value will have to be doubled. This is illustrated below.



In addition, we also add edges $(s, \rho(v), v)$ and $(\tilde{v'}, \rho(v'), t)$ to E'. We now get the automaton $\mathcal{A}_{G,v,v'}$ as the automaton obtained from G' by setting s as the initial and t as the final state. Note that the construction of the automaton $\mathcal{A}_{G,v,v'}$ is polynomial.

4. Finally, for $Q = x \xrightarrow{e} y$ we have $(v,v') \in Q(G)$ iff the language $\mathcal{A}_{G,v,v'}$ has nonempty intersection with the language generated by the grammar $\mathcal{G}_{e,D}$. This follows by an argument similar to the proof of Proposition 4.1.5.

   Since the intersection of a context-free language and a regular language is context-free and can be obtained by the product construction of a PDA and an NFA, this means that $(v,v') \in Q(G)$ iff the product $\mathcal{A}(\mathcal{G}_{e,D}) \times \mathcal{A}_{G,v,v'}$ defines a nonempty language. This product is a PDA, so we can check its nonemptiness in polynomial time, giving us a polynomial algorithm for query evaluation.

Steps 2, 3, and 4 above use the standard constructions of converting CFGs into PDAs, taking products, and checking PDAs for nonemptiness. So what is missing is the construction of the CFG $\mathcal{G}_{e,D}$, which we show next.

Regular expressions with equality into CFGs Assume that we have a finite set D of data values. We now inductively construct CFGs $\mathcal{G}_{e,D}$ for all regular expressions with equality. The terminal symbols of these CFGs will be Σ plus all elements of D. All nonterminals in $\mathcal{G}_{e,D}$ will be of the form $A_{e'}$ and $A_{e'}^{dd'}$, where e' ranges over subexpressions of e and $d,d' \in D$. Intuitively, words derived from $A_{e'}^{dd'}$ will correspond to (in a way previously described) data paths in $L(e')$ with data values from D that start with d and end with d'; words derived from $A_{e'}$ will correspond to data paths in $L(e')$ with data values from D. The start symbol for the grammar corresponding to the expression e will be A_e.

The productions of the grammars $\mathcal{G}_{e,D}$ are now defined inductively as follows.

- If $e = \varepsilon$, we have productions $A_\varepsilon \to \bigvee_{d \in D} A_\varepsilon^{dd}$ and $A_\varepsilon^{dd} \to d$ for each $d \in D$.
- If $e = a$, for $a \in \Sigma$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to dad'$ for all $d,d' \in D$.
- If $e = e_1 \cdot e_2$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to \bigvee_{d'' \in D} A_{e_1}^{dd''} A_{e_2}^{d''d'}$ for all $d,d' \in D$ together with all the productions of the grammars $\mathcal{G}_{e_1,D}$ and $\mathcal{G}_{e_2,D}$.
- If $e = e_1 + e_2$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'} | A_{e_2}^{dd'}$ for all $d,d' \in D$ together with all the productions of the grammars $\mathcal{G}_{e_1,D}$ and $\mathcal{G}_{e_2,D}$.
- If $e = (e_1)^+$, we have productions $A_e \to \bigvee_{d,d' \in D} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'} | \bigvee_{d'' \in D} A_{e_1}^{dd''} A_e^{d''d'}$ for all $d,d' \in D$ together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.
- If $e = e_1[c]$, we have productions $A_e \to \bigvee_{d,d' \in D, d' \models c} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'}$ for all $d,d' \in D$ where $d' \models c$, together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.

- If $e = (e_1)_{=}$, we have productions $A_e \to \bigvee_{d \in D} A_e^{dd}$ and $A_e^{dd} \to A_{e_1}^{dd}$ for all $d \in D$ together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.
- If $e = (e_1)_{\neq}$, we have productions $A_e \to \bigvee_{d,d' \in D,\ d \neq d'} A_e^{dd'}$ and $A_e^{dd'} \to A_{e_1}^{dd'}$ for all $d, d' \in D$ with $d \neq d'$, together with all the productions of the grammar $\mathcal{G}_{e_1,D}$.

It is clear from the construction that all words generated by this grammar(with the sole exception of the empty word) have all of their intermediate data values (i.e. letters corresponding to values in D) doubled, except the first and the last one.

Note that with these expressions we assume that ε can appear only when denoting the empty word and will be removed otherwise. We require this, so that we would not get productions that produce objects that are not data paths, such as e.g. ddd for the expression $\varepsilon \cdot \varepsilon \cdot \varepsilon$. Note that this is not a problem, since all expressions can be rewritten to be of this form in DLOGSPACE.

The main result connecting these CFGs with languages of regular expressions with equality is this. Recall that when we say that a word over Σ and D corresponds to a data path with values in D, we mean that it equals the data path with all the data values, except the first and the last, doubled.

Proposition 4.4.7. *The language of words derived by each CFG $\mathcal{G}_{e,D}$ corresponds to the set of data paths in $L(e)$ whose data values come from D. Furthermore, the set of words derived from each nonterminal $A_e^{dd'}$ corresponds to the set of data paths in $L(e)$ which start with d, end with d', and whose data values come from D.*

Moreover, the CFG $\mathcal{G}_{e,D}$ can be constructed in polynomial time from e and D.

Proof. We prove the proposition by induction on the structure of e. Note that it is enough to show the second claim, i.e. we will show that the set of words derived from each nonterminal $A_e^{dd'}$ corresponds to the set of data paths in $L(e)$ which start with d, end with d', and whose data values come from D. This means that a word $d_1a_1d_2d_2a_2d_3d_3 \ldots a_{n-1}d_n$ in which all values but first and last are doubled is derived from $A_e^{dd'}$ if and only if data path $d_1a_1d_2a_2d_3 \ldots a_{n-1}d_n$ is in $L(e)$ and uses data values from D. We prove this by induction on the structure of the expression.

- If $e = \varepsilon$, or $e = a$, with $a \in \Sigma$, the claim is immediate.
- If $e = e_1 + e_2$ then $A_e^{dd'} \to A_{e_1}^{dd'} | A_{e_2}^{dd'}$. But then each word in $A_e^{dd'}$ is either in $A_{e_1}^{dd'}$ or in $A_{e_2}^{dd'}$, so the claim follows from the induction hypothesis.
- If $e = e_1 \cdot e_2$, we have a production $A_e^{dd'} \to \bigvee_{d'' \in D} A_{e_1}^{dd''} A_{e_2}^{d''d'}$. To see the equivalence assume first that w is generated by $A_e^{dd'}$. This means that there exists $d'' \in D$ such that w is generated by $A_{e_1}^{dd''} A_{e_2}^{d''d'}$. By definition this means that $w = w_1 \cdot w_2$ such that w_1 is generated by $A_{e_1}^{dd''}$ and w_2 is generated by $A_{e_1}^{d''d'}$. By the induction hypothesis this implies that data path w_1' corresponding to w_1, is in the language of e_1, starts with d and ends

with d''. Likewise w'_2, a data path corresponding to w_2 starts with d'', ends with d' and is in the language of e_2. Note that the induction hypothesis also implies that the splitting of the word is correct. Since w'_1 ends with d'' and w'_2 begins with it we can concatenate these two data paths to get w', a data path corresponding to w, that is in the language of e, begins with d and ends with d' as required.

Conversely, suppose that $w' \in L(e)$ is a data path that begins with d, ends with d' and takes only data values from the set D. By definition of concatenation there exists a splitting $w' = w'_1 \cdot w'_2$ such that $w'_1 \in L(e_1)$ and $w'_2 \in L(e_2)$. Since w' takes data values from D there is some d'' such that w'_1 ends with d'' and w'_2 begins with d''. But then by the induction hypothesis w_1, word obtained from w'_1 by doubling all intermediate data values, will be generated by $A_{e_1}^{dd''}$, while w_2, a word obtained from w'_2 by doubling all intermediate data values, will be generated by $A_{e_2}^{d''d'}$. But then their concatenation $w = w_1 \cdot w_2$ is precisely the word corresponding to data path w' and is generated by $A_{e_1}^{dd''}A_{e_2}^{d''d'}$ and thus $A_e^{dd'}$.

- If $e = (e_1)^+$, we have a production $A_e^{dd'} \to A_{e_1}^{dd'} \mid \bigvee_{d'' \in D} A_{e_1}^{dd''}A_e^{d''d'}$. This implies that every word is generated either by $A_{e_1}^{dd'}$, in which case the claim follows immediately from the induction hypothesis, or is generated by $\bigvee_{d'' \in D} A_{e_1}^{dd''}A_e^{d''d'}$, in which case the proof mimics the proof for the concatenation case, taking into account that recursion will terminate after finitely many steps and thus the final expression will be a multiple concatenation of terms for which the induction hypothesis holds.
- If $e = e_1[c]$, we have $A_e^{dd'} \to A_{e_1}^{dd'}$, where $d' \models c$, which by the induction hypothesis corresponds to all words in $L(e)$ with data values from D.
- If $e = (e_1)_=$, we have $A_e^{dd} \to A_{e_1}^{dd}$, which by the induction hypothesis corresponds to all words in $L(e)$ with data values from D.
- If $e = (e_1)_{\neq}$, we have $A_e^{dd'} \to A_{e_1}^{dd'}$, where $d \neq d'$, which by the induction hypothesis corresponds to all words in $L(e)$ with data values from D.

To see that the grammar for an expression e can be constructed in polynomial time observe that there are at most $O(n^2)$ subexpressions of e, where the length of e is n. Since the grammar for e is constructed by starting from subexpressions and taking unions of already constructed subgrammars and every new rule adds at most $O(|D|^3)$ productions to our grammar we get a grammar of the size at most $O(n^2 \cdot |D|^3)$. Note that we reuse old subgrammars so we do not get exponential blow-up. □

This, together with the algorithm shown above, finally gives us tractability of combined complexity.

Theorem 4.4.8. *Combined complexity of RQD queries is in* PTIME.

Proof. It is clear from the description that algorithm runs in polynomial time. It remains to prove that it is correct, i.e. that for $Q = x \xrightarrow{e} y$ we have $(v,v') \in Q(G)$ iff the language of $\mathcal{A}_{G,v,v'}$ has nonempty intersection with the language generated by $\mathcal{A}(\mathcal{G}_{e,D})$.

To see this assume first that $(v,v') \in Q(G)$. This means that there is a data path w_π form v to v' in G such that $w_\pi \in L(e)$. By Proposition 4.4.7 this implies that the corresponding word with all intermediate data values doubled is in the language of $\mathcal{G}_{e,D}$ and thus $\mathcal{A}(\mathcal{G}_{e,D})$. Also, since w_π is a path in G it is of the form $d_1 a_1 \dots a_{n-1} d_n$, where $d_i = \rho(v_i)$, for $i = 1,\dots,n$, for some nodes $v_1,\dots,v_n$ in G such that $v_1 = v$ and $v_n = v'$. This implies that (v_i, a_i, v_{i+1}) is an edge in E, for $i = 1,\dots,n-1$. This again implies that $a_i d_{i+1} d_{i+1}$ enables us to change the state of $\mathcal{A}_{G,v,v'}$ from v_i to v_{i+1} (by going through $\tilde{v}_{i+1}$ and $\hat{v}_{i+1}$), for $i = 2,\dots,n-1$. Since (s, d_1, v_1) and $(\tilde{v}_n, d_n, v_n)$ are also transitions in $\mathcal{A}_{G,v,v'}$ (as well as $(v_{n-1}, a_{n-1}, \tilde{v}_n)$) we see that $\mathcal{A}_{G,v,v'}$ accepts the word $d_1 a_1 d_2 d_2 a_2 d_3 d_3 \dots a_{n-1} d_n$, i.e. the word corresponding to w_π. It follows that the intersection of $\mathcal{A}(\mathcal{G}_{e,D})$ and $\mathcal{A}_{G,v,v'}$ is nonempty.

Conversely, assume that the product $\mathcal{A}_{G,v,v'} \times \mathcal{A}(\mathcal{G}_{e,D})$ defines a nonempty language and that $w' = d_1 a_1 d_2 d_2 a_2 d_3 d_3 \dots a_{n-1} d_n$ is some word in that language. If we delete doubled data values from w' (remember the discussion before the statement of Proposition 4.4.7 where we show that all words in $L(\mathcal{G}_{e,D})$ are of this form) we get a word w. By Proposition 4.4.7, w will be in the language of e. On the other hand, since $w' \in L(\mathcal{A}_{G,v,v'})$ we know that there is a run from s to t in $\mathcal{A}_{G,v,v'}$ that accepts this word. Then by the construction of this automaton there exists a sequence $v_1,\dots,v_n$ of nodes from G such that $d_i = \rho(v_i)$ are the appropriate data values, $(v_i, a_i, v_{i+1}) \in E$ the corresponding edges and $v = v_1$, while $v' = v_n$. It is clear that w coincides with data path defined by this path and is thus a data path in G starting in v and ending in v'. We conclude that $(v,v') \in Q(G)$. □

We also note that a simpler dynamic programming algorithm that evaluates *RQD*s bottom-up can be applied to prove membership in PTIME. We will describe this algorithm in Section 7.1 where it will be used to evaluate queries from a more expressive language called GXPath. We have opted for the approach taken here to emphasise connection with formal languages.

4.5 Variable automata

We have seen in previous sections that query languages tend to have either polynomial or PSPACE combined complexity when evaluated on graph databases. A natural question to ask is if we can find a reasonable formalism whose combined complexity will be between these two classes.

Here we do so by using variable automata introduced in [Grumberg et al., 2010a]. These automata can be viewed as less procedural than register automata; in fact they can be seen as NFAs with a guess of values to be assigned to variables, with the run of the automaton verifying

correctness of the guess. Originally they were defined on words over infinite alphabets [Grumberg et al., 2010a], but it is straightforward to extend them to the setting of data graphs. In what follows we define variable automata as a formalism for defining languages of data paths and show how they can be used to post queries on graph databases. We will also give several examples of such queries and show that they can be evaluated in NP-time with respect to combined complexity.

We begin by defining variable automata formally.

Definition 4.5.1. *Let Σ be a finite alphabet and $\mathcal{D}$ an infinite domain of data values. We will also assume that we have a countable set V of variables. A* variable finite automaton *(or VFA for short) over Σ is a a tuple $\mathcal{A} = (Q, q_0, F, \Gamma, \delta)$, where:*

- *$Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;*
- *$q_0 \in Q_d$ is the initial state;*
- *$F \subseteq Q_w$ is the set of final states;*
- *$\Gamma = C \cup X \cup \{\star\}$ such that:*
  - *$C \subseteq \mathcal{D}$ is a finite set of data values called* constants
  - *$X \subseteq V$ is a finite set of* bound variables*, and*
  - *$\star$ is a symbol for the* free variable.
- *$\delta = (\delta_w, \delta_d)$ is a pair of transition relations:*
  - *$\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;*
  - *$\delta_d \subseteq Q_d \times \Gamma \times Q_w$ is the data transition relation.*

Next we define when a VFA $\mathcal{A}$ accepts a data path $w = d_0 a_0 d_1 a_1 \dots d_n a_n d_{n+1}$.

Let $v = v_0 b_0 v_1 b_1 \dots v_n b_n v_{n+1}$ be a word where $v_0, \dots v_{n+1} \in \Gamma$ and $b_0, \dots b_n \in \Sigma$. We will say that v is a *witnessing pattern* for w (or that w is a *legal instance* of v) if there is a sequence $q_0, q'_0, q_1, q'_1 \dots q_n, q'_n, q_{n+1}, q'_{n+1}$ of states in $\mathcal{A}$, with $q'_{n+1} \in F$ such that the following holds:

1. for each i we have $(q_i, v_i, q'_i) \in \delta_d$ and $(q'_i, b_i, q_{i+1}) \in \delta_w$,
2. $a_i = b_i$ and $(q'_i, a_i, q_{i+1}) \in \delta_w$, for $i = 0, \dots, n$,
3. if $v_i = c \in C$ then $(q_i, c, q'_i) \in \delta_d$ and $d_i = c$,
4. if $v_i, v_j \in X$ then $d_i = d_j$ iff $v_i = v_j$ and $d_i, d_j \notin C$,
5. if $v_i = \star$ and $v_j \neq \star$ then $d_i \neq d_j$.

Intuitively the definition states that in a legal instance constants and finite alphabet part will remain unchanged (conditions 2 and 3), while every bound variable is assigned with the

same *unique* data value from $\mathcal{D} - C$ (condition 4) and every occurrence of the free variable $\star$ is freely assigned any data value from $\mathcal{D} - C$ that is not assigned to any of the bound variables (condition 5). Note that the condition 5 is a lot stronger that saying that $\star$ means any data value.

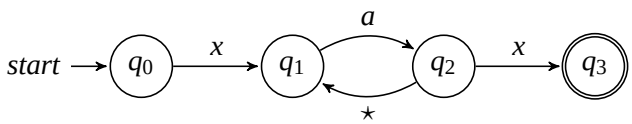
Intuitively, finding a witnessing pattern for a data path is the same as guessing an assignment which maps each constant, bound, or free variable to an appropriate data value in the path. This assignment is then checked against conditions above to make sure all data value comparisons are as specified by the automaton. One important property of condition 5 though, is that unlike all the other ones, it is not dependant only on the current and next state of the automaton, but allows it to reason along the whole run.

We now define the *language of* $\mathcal{A}$, or simply $L(\mathcal{A})$ for short, as the set of all data paths w for which there exists a witnessing pattern v.

Note that it is straightforward to define regular-like expressions for VFAs that will simply inherit the associated semantics.

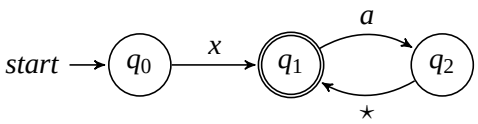
Example 4.5.2. *Here we give a few examples of languages accepted by VFAs.*

1. *The language where the first data value is equal to the last and all other values are different from them (but can be equal among themselves).*



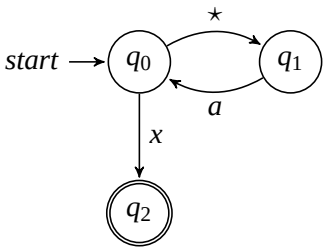
   The witnessing patterns here has the form $x(a \cdot \star)^* \cdot a \cdot x$, so condition 5 will imply that the data values in the middle are different from the first and the last.

2. *The language where the first data value is different from all other data values.*



   This time the witnessing pattern takes the form $x(a \cdot \star)^*$, thus dictating that the first data value is never repeated.

3. *The language where the last data value differs from all other data values.*



Finally in this example, the witnessing pattern has the form $(\star \cdot a)^* x$, so the last value can never be replicated because of the condition 5.

Note that the last example is not expressible by register automata [Kaminski and Francez, 1994].

It was shown in [Grumberg et al., 2010b] that the language $L = \{d_1 a d_1 a d_2 a d_2 a \ldots d_k a d_k a d_{k+1} \mid k \geq 1\}$ is not expressible by VFAs. However, it is straightforward to show that this language is defined by the regular expression with equality $((a)_= \cdot a)^+$. Thus, we obtain:

Proposition 4.5.3. *VFAs are incomparable in terms of expressive power with register automata, regular expressions with binding and regular expressions with equality.*

Regular queries with variables

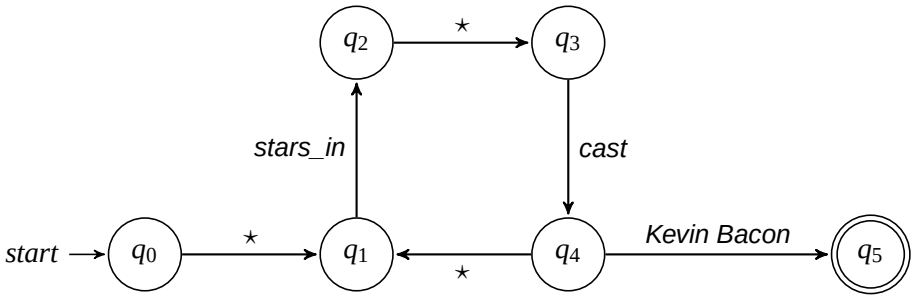
Here we define a class of queries based on variable automata and examine the complexity of their query evaluation problem.

Definition 4.5.4. *A* regular query with variables *is an expression $Q = x \xrightarrow{\mathcal{A}} y$, where $\mathcal{A}$ is a variable automaton.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(\mathcal{A})$.

The class of these queries is denoted by RQV.

Example 4.5.5. *Coming back to the database in Figure 2.3, we can use the following variable automaton $\mathcal{A}$ to specify a query returning all the actors who have a finite Bacon number:*



As before, we also return the node corresponding to Kevin Bacon due an inherent limitation of path languages to define unary queries. Note that the automaton $\mathcal{A}$ does not allow Kevin Bacon to appear more than once along a path due to condition 5 in Definition 4.5.1. This, however, does not affect the intended semantics of our query.

As announced we can now prove that for queries posted by variable automata combined complexity of query evaluation drops to *NP*. Moreover, we also show the matching lower bound.

Theorem 4.5.6. *Combined complexity of query evaluation problem for RQVs is NP-complete.*

Proof. First we prove membership. Assume we are given a graph G, two nodes $s,t \in G$ and a *RQV* Q specified by a VFA $\mathcal{A}$. We show that if π is a path in G from s to t, such that $w_\pi \in L(\mathcal{A})$ then there is also a path π' in G from s to t of length at most $n := |G+1| \cdot (2|\mathcal{A}|+1)+1$ such that $w_{\pi'} \in L(\mathcal{A})$, where $|\mathcal{A}|$ denotes the number of states in $\mathcal{A}$.

Assume that $\pi = n_0, \ldots, n_{l+1}$ is a path of length greater than n such that $n_0 = s, n_l = t$ and the associated data path $w_\pi = d_0 a_0 \ldots d_l a_l d_{l+1}$ belongs to the language $L(\mathcal{A})$. Let $v = v_0 b_0 \ldots v_l b_l v_{l+1}$ be a witnessing pattern for w_π. Then there is a sequence $q_0, q'_0, q_1, q'_1, \ldots, q_{l+1}, q'_{l+1}$ of states of $\mathcal{A}$ that confirms this according to definition of acceptance by VFAs.

By the pigeon hole principle there exists $i, j \leq l$ such that $n_i = n_j$ and $q_i = q_j$. Observe that $\pi' = n_0, \ldots, n_i, n_{j+1}, \ldots, n_{l+1}$ is still a path in G from s to t with the associated data path $w' = d_0 a_0 \ldots a_{i-1} d_j a_j d_{j+1} \ldots d_l a_l d_{l+1}$ and that $v' = v_0 b_0 \ldots b_{i-1} v_j b_j v_{j+1} \ldots v_l b_l v_{l+1}$ is a witnessing pattern for w', as verified by the sequence $q_0, q'_0, \ldots q_i, q'_j q_{j+1}, \ldots q'_{l+1}$ of states.

By repeating this cutting procedure we get the desired result. Now for the NP-algorithm we simply guess a path of length at most n, which is polynomial in the size of the input, and verify that it belongs to our language in PTIME. Note that in order to obtain an *NP* algorithm we also guess an assignment of data values to variables in our expression at the same time as guessing the path (thus effectively guessing the witnessing pattern). This is necessary since membership for VFAs is *NP*-complete [Grumberg et al., 2010a].

To show NP-hardness we do a reduction from k-CLIQUE. This problem asks, given a graph G and a number k, to determine if G has a clique of size at least k.

Suppose we are given an undirected graph G and a number k. We will construct a graph G' with $|G|+2$ nodes , select two nodes $s,t \in G'$ and construct a VFA $\mathcal{A}$ of size $O(k^2)$ such that G contains a k-clique if and only if there is a path from s to t in G' whose associated data path belongs to $L(\mathcal{A})$.

Take $\Sigma = \{a,b\}$ and make G directed by adding edges in both directions for every edge in G. Assume that every vertex v is given an unique data value d_v. Label the edges $(v,v') \in G'$ by a and add two more nodes s and t, with unique values d_s and d_t attached. Add an edge from s to every other node v except s,t and label them with b. Also add an edge from every node in G to t and label them by b. We call the resulting graph G'. (The idea is that every node has a unique data value – its id.)

We define our VFA as a linear path with transitions:

- $(q_0, d_s, q'_0), (q'_0, b, q_1), (q_1, x_1, q'_1), (q'_1, a, q_2), (q_2, x_2, q'_2)$ (this collect the first two nodes in the clique),

- $(q'_{i-1}, a, q_i), (q_i, x_i, q'_i)$, (selecting node i)
  $(q'_i, a, q^i_1), (q^i_1, x_1, \overline{q^i_1}), (\overline{q^i_1}, a, p^1_i), (p^1_i, x_i, \overline{p^1_i})$, (checking it is connected with the first node selected)
  $(\overline{p^1_i}, a, q^i_2), (q^i_2, x_2, \overline{q^i_2}), (\overline{q^i_2}, a, p^2_i), (p^2_i, x_i, \overline{q^i_2}), \ldots$, (checking it is connected with the second node)
  $(\overline{p^{i-2}_i}, a, q^i_{i-1}), (q^i_{i-1}, x_{i-1}, \overline{q^i_{i-1}})$,
  $(\overline{q^i_{i-1}}, a, q_i), (q_i, x_i, q'_i)$ ((checking it is connected with the last node selected)), for $3 \leq i \leq k$ and
- $(q'_k, b, q_{k+1}), (q_{k+1}, d_t, q'_{k+1})$ (to get the target node).

Note that here we add a new state for each transition of the automaton.

Next we show that there is a k-clique in G iff there is a data path form s to t in G' whose label belongs to $L(\mathcal{A})$.

Suppose first that there is a k-clique in G. Then we simply move from s to arbitrary point in that clique using the b labelled edge and traverse the clique back and forth until we reach the k-th element of the clique. Note that starting from the third element, whenever we select a different node in the clique we have to move back and forth between this node and all previously selected ones to match the transitions (we check that they are interconnected), but since we have a clique this is possible. Finally, after selecting the last node and verifying that it is connected to all the others we move to t using a b labelled edge.

Now suppose that there is a path from s to t in G' whose label belongs to $L(\mathcal{A})$. This means that we will be able to select k different nodes $n_1, \ldots, n_k$ in G with data values stored in $x_1, \ldots, x_k$. Since all data values in the graph are different they also act as ids. Now take any two n_l, n_m with $l < m \leq k$. Then we know that n_l and n_m are connected in G because after selecting n_m we have to go through the transitions stating $(\overline{p^{l-1}_m}, a, q^m_l), (q^m_l, x_l, \overline{q^m_l}), (\overline{q^m_l}, a, p^l_m), (p^l_m, x_m, \overline{p^l_m})$ and similarly for when l, m are at the beginning or end of the transition chain. Since no two data values in G are the same this means that we have an edge between n_l and n_m. This completes the proof. □

Furthermore, we can also show that data complexity remains in NL.

Proposition 4.5.7. *Data complexity for RQV queries is NL-complete.*

Proof. Assume that we have a fixed query Q specified by a VFA $\mathcal{A}$. We are given G and $s, t \in G$ as input. Using the same construction as in the proof of Theorem 4.1.6 we can transform the graph G into a graph G' with a number of nodes doubled. Note that this G' can be viewed as a VFA that uses only constants and with s_s as initial and t_t as the final state (these nodes correspond to v_s and v_t in the aforementioned construction).

Using Theorem 1 in [Grumberg et al., 2010a] we build the product of our graph G', viewed as a VFA and our fixed VFA $\mathcal{A}$. Theorem 2 in [Grumberg et al., 2010b] counts the number of

states in the product construction as $O(n_1 \cdot n_2 \frac{(d_1+d_2+c_1+c_2)!}{(c_1+c_2)!})$ and the number of transitions as $O(\frac{(d_1+d_2+c_1+c_2)!}{(c_1+c_2)!})$, where n_i is the number of states, d_i the number of bounded variables and c_i the number of constants from $\mathcal{D}$ in each of our automatons.

Note now that since $\mathcal{A}$ is fixed n_2, d_2 and c_2 are constants. Let $M = n_2 + d_2 + c_2$. Also notice that our graph, viewed as an automaton has $d_1 = 0$ and n_1 and c_1 are both bounded by the size of the graph $|G|$. Thus the size of our product automaton is $O(M \cdot |G| \frac{(M+|G|)!}{(c_1+|G|)!}) \leq O(M \cdot |G| \cdot (M+|G|)^M)$, that is polynomial in the size of G and the same calculation applies to the number of transitions.

Using standard on-the-fly technique we check the product automaton for nonemptiness in NL. It is straightforward to see that (s,t) is in the answer to our query Q on G if and only if this product is nonempty. Thus we get the desired upper bound.

Lower bound follows from the same result for RPQs (without data values). □

Note that the combined complexity dropped from PSPACE to NP, which is viewed as much more acceptable for query evaluation, at least over large databases. This is the complexity of relational conjunctive queries, for instance [Abiteboul et al., 1995], or conjunctive regular path queries over graphs [Consens and Mendelzon, 1990].

4.6 Summary of complexity results

We have seen in previous sections that even when the most expressive class of queries, those based on register automata, are considered, the combined complexity matches that of usual relational calculus queries [Abiteboul et al., 1995], or RPQs extended with rational relations [Barceló et al., 2012b]. Data complexity, on the other hand, is the best possible in light of the results for RPQs (which basically follow from the bounds for graph reachability problem [Jones, 1975]). These results extend to the class of RQMs and even to RQBs, which restrict automata and RQMs with proper scoping rules, making them slightly weaker, but closer in syntax to usual programming languages. When expressions are further restricted we arrive at the class of RQDs, whose combined complexity drops to PTIME. From this we can see that there is somewhat of a split between path languages when combined complexity of the query evaluation problem is considered. Namely it it is rather PSPACE or PTIME. In our search for a formalism with intermediate complexity we showed that when queries are based on variable automata one indeed gets an NP bound. All of these results are summarised in Table 4.1.

Query evaluation	RDPQ	RQM	RQB	RQD	RQV
combined complexity	PSPACE-c	PSPACE-c	PSPACE-c	PTIME	NP-c
data complexity	NL-c	NL-c	NL-c	NL-c	NL-c

Table 4.1: Complexity of the query evaluation problem

Chapter 5

Additional features

An important issue in query language design is enriching the base theoretical languages with features required from database practitioners. In the context of graph databases two of the most important such features are the ability to traverse edges backwards and allowing conjunctive queries to be formed from simple graph queries. Indeed, it has been argued before [Calvanese et al., 2000, Calvanese et al., 2003] that the inverse operator is a required feature of any practical graph language, while the usefulness of conjunctive queries has been well studied both on relational databases [Abiteboul et al., 1995] and on graphs [Barceló et al., 2012b, Freydenberg and Schweikardt, 2011, Bienvenu et al., 2013].

In this chapter we will first examine what happens when path languages from Chapter 4 are enriched with the inverse operator. Here we will take the approach somewhat different from the one in that chapter and define our queries to work directly on graphs, instead of taking the additional detour through language theory. This will allow us to obtain a uniform semantics for all classes of queries and to define inverse operators in a simple way. We will also show that the two semantics are equivalent and that enriching our languages with the ability to traverse graph edges in both ways has no impact on the complexity of the query evaluation problem.

Following that, we will study the impact of conjunction on languages from Chapter 4, showing that in most cases no cost is incurred, except when lower bounds are already dictated by weaker classes of queries. In particular we can obtain optimal query evaluation bounds, in light of those for single queries.

Finally, we will also show that by merging two incomparable formalisms from the previous chapter, that of register automata and variable automata, we can obtain a highly expressive model with no incurred cost in evaluation complexity. However, as we argue at the end of the chapter, such a model requires much care when designing queries and is thus highly unlikely to be adopted as a querying standard for data graphs.

5.1 Languages with inverse

All of the languages considered in the previous chapter can be viewed as extensions of RPQs which manage data values. However, as noted in [Calvanese et al., 2000], RPQs by themselves lack a very natural construction for navigation through the structure of graphs—namely, the *inverse* operator. Indeed, consider for example a genealogy graph over a single *parent* label, such as the one presented in the following figure.

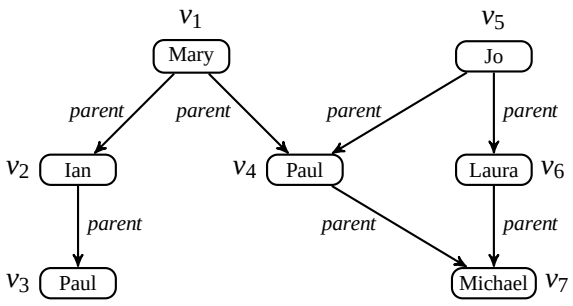


Figure 5.1: A genealogy database over the *parent* label.

We assume that nodes represent people and data values are their names. A natural query over this graph, which does not deal with data values, would be to ask for all pairs of siblings. This, however, is clearly not expressible as an RPQ. On the other hand, it can be written as $parent^- parent$, where '$^-$' is the inverse operator, which traverses edges *backwards*. This query will retrieve e.g. (v_2, v_4) from the graph in Figure 5.1, since these nodes have a common parent v_1.

The class of queries enriching RPQs with inverse, called *2-way RPQs*, or *2RPQs* for short, was introduced in [Calvanese et al., 2000], where it was shown that even with this extension query evaluation remains the same as for RPQs (namely NLOGSPACE-complete). Moreover, in [Calvanese et al., 2003] the authors also show that query containment is as efficient as for plain RPQs (namely PSPACE-complete).

Here we will consider the extensions of queries defined in Chapter 4 with the inverse operator. As argued above such extensions are natural from a navigational point of view, but they can also be used to ask interesting queries where data values are involved and should thus be incorporated into formalisms for querying data graphs. For example, one query of interest in our genealogy database might be to retrieve all pairs of (blood) relatives with the same name. This can be easily done by the means of two-way RQD $((parent^-)^+ parent^+)_=$, which checks that two people have a common ancestor and ensures that they also have the same name. For example the pair (v_3, v_4) is an answer to this query in our sample graph. Next we define this class of queries formally.

Graph semantics

As mentioned in Section 2.5, semantics of regular path queries can be defined directly on graphs, without taking a detour through language theory. Here we show that the same can be done for the classes of queries based on expressions with memory, binding and equality. In this respect we can identify e.g. regular expressions with memory and regular queries with memory and say that an expression e is an *RQM* and vice versa (recall the discussion in Remark 2). As demonstrated before, this approach will allow us to have a uniform relational semantics for all the languages we consider, as well as allowing us to bypass a somewhat awkward approach using semi-paths from [Calvanese et al., 2000] when defining the inverse operator. Note that here we will not consider register nor variable automata with inverses, as such model amounts to more than simply adding the ability to traverse edges of a graph in both directions and has some deeper language theoretic implications which would detract us from our goal to study languages for querying graph databases.

Two-way regular queries with memory (2*RQM*s).

Here we will define the language of 2*RQM*s to work directly on graph databases, thus removing the distinction between (two way) regular expressions with memory and 2*RQM*s. In that respect we will from now on identify the two and say that a two-way regular expression with memory e is an 2*RQM* and vice versa. We will also show that this approach is equivalent to the one taken in Section 4.2.

The syntax of 2*RQM*s is defined by extending Definition 4.2.1 with the inverse operator. That is, for a finite alphabet Σ and a set $\{x_1,\ldots,x_k\}$ of variables, they are expressions specified by the following grammar:

$$e \;:=\; \varepsilon \mid a \mid a^- \mid e+e \mid e\cdot e \mid e^+ \mid e[c] \mid \downarrow\bar{x}.e \tag{5.1}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\bar{x}$ over tuples of variables from $x_1,\ldots,x_k$.

To define semantics of 2*RQM*s we will need some additional terminology. Given a data graph G and a set of variables X, a *state* is a pair consisting of a node of G and an assignment of variables $\sigma : X \to \mathcal{D}$.

The semantics of 2RQMs over a data graph $G = \langle V, E, \rho\rangle$ can then be defined in terms of function $\mathcal{H}^G$, which associates with each 2RQM a set of pairs of states. The intuition of the set $\mathcal{H}^G(e)$, for some 2RQM e, is as follows. Given states $s = (v,\sigma)$ and $s' = (v',\sigma')$, the pair (s,s') is in $\mathcal{H}^G(e)$ if there exists a path w from v to v', such that the expression e can parse w assuming that the variables are initialized according to σ, modified and compared as dictated by e, and the resulting assignment after traversing the path is σ'.

Formally, given a data graph $G = \langle V, E, \rho \rangle$, the function $\mathcal{H}^G$ is constructed by the following inductive definition.

$$
\begin{array}{lcl}
\mathcal{H}^G(\varepsilon) & = & \{(s,s) \mid s \text{ is a state}\}, \\
\mathcal{H}^G(a) & = & \{((v,\sigma),(v',\sigma)) \mid (v,a,v') \in E\}, \\
\mathcal{H}^G(a^-) & = & \{((v',\sigma),(v,\sigma)) \mid (v,a,v') \in E\}, \\
\mathcal{H}^G(e_1 \cup e_2) & = & \mathcal{H}^G(e_1) \cup \mathcal{H}^G(e_2), \\
\mathcal{H}^G(e_1 \cdot e_2) & = & \mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2), \\
\mathcal{H}^G(e^+) & = & \mathcal{H}^G(e) \cup \mathcal{H}^G(e \cdot e) \cup \ldots, \\
\mathcal{H}^G(e[c]) & = & \{((v,\sigma),(v',\sigma')) \mid ((v,\sigma),(v',\sigma')) \in \mathcal{H}^G(e) \text{ and } \rho(v'),\sigma' \models c\}, \\
\mathcal{H}^G(\downarrow x.e) & = & \{((v,\sigma),(v',\sigma')) \mid ((v,\sigma),(v',\sigma')) \in \mathcal{H}^G(e) \text{ and } \sigma(x) = \rho(v)\}.
\end{array}
$$

The symbol $\circ$ above refers to the usual composition of binary relations:

$$\mathcal{H}^G(e_1) \circ \mathcal{H}^G(e_2) = \{(s_1,s_3) \mid \exists s_2 \text{ s.t. } (s_1,s_2) \in \mathcal{H}^G(e_1) \text{ and } (s_2,s_3) \in \mathcal{H}^G(e_2)\}.$$

Finally, the *evaluation* $[\![e]\!]^G$ of an 2*RQM* e over a data graph G is the following set of pairs of nodes in G:

$$\{(v,v') \mid \exists \sigma' \text{ s.t. } ((v,\perp),(v',\sigma')) \in \mathcal{H}^G(e)\},$$

where $\perp$ is the empty assignment.

To see that 2*RQM*s indeed extend *RQM*s from Section 4.2 we have to show that when the language without the inverse operator is considered, the semantics given here matches the one for regular queries with memory defined in Section 4.2.

Proposition 5.1.1. *Let e be any regular expression with memory and $Q = x \xrightarrow{e} y$ an RQM. Then for any data graph G it holds that $(v,v') \in [\![e]\!]^G$ if and only if $(v,v') \in Q(G)$.*

Proof. Note first that any regular expression with memory is also a 2*RQM* (for this see Grammar 5.2). Proof can now be carried out by a routine induction on the structure of e. □

It is important to note that two things can be inferred from this:

a) With graph semantics for 2RQMs we can avoid defining two-way queries using semi-paths.

b) This also gives us graph semantics for RQMs, as the previous proposition illustrates.

Furthermore, we can also show that graph semantics for *RQM*s can be used to define data path languages in a way that is equivalent to Definition 4.2.1. For that note first that every data path $w = d_1 a_1 d_2 \ldots a_{n-1} d_n$ can be easily transformed to a data graph G_w, consisting of n different nodes with data values d_1, …, d_n, respectively, consequently connected by edges labelled with $a_1, \ldots, a_{n-1}$, as illustrated in the following figure.

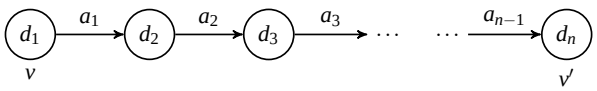


Figure 5.2: Data graph corresponding to the data path w

We could then say that a data path w is accepted by regular expression with memory e if and only if $(v,v') \in [\![e]\!]^{G_w}$. That this is equivalent to Definition 4.2.1 follows from the lemma below.

Lemma 5.1.2. *Take any regular expression with memory e and a data path w. Then for any two assignments* σ, σ' *it holds that*

$$(e, w, \sigma) \vdash \sigma' \iff ((v,\sigma),(v',\sigma')) \in \mathcal{H}^{G_w}(e).$$

Proof. This can be easily shown by a straightforward induction on the structure of expression e. □

We could thus also define the language of data paths accepted by regular expressions with memory using this graph semantics: a data path w is *accepted* by e iff $(v,v') \in [\![e]\!]^{G_w}$, where v and v' are the first and the last node of G_w. This shows that the two definitions are indeed dual when one-way languages are considered.

Next we show that even with this additional functionality the same complexity of query evaluation applies to 2*RQM*s as does to their one-way variant.

Proposition 5.1.3. *The problem of deciding whether a pair of nodes belongs to* $[\![e]\!]^G$ *for a 2RQM e and a data graph G is* PSPACE*-complete. If we assume that e is fixed the problem becomes* NLOGSPACE*-comlete.*

Proof. Take any 2RQM e over Σ and a data graph G. Let $\Sigma' = \Sigma \cup \{a^- : a \in \Sigma\}$ and let $G' = \langle V, E', \rho \rangle$, where V and ρ are as in G, while $E' = E \cup \{(v', a^-, v) : (v, a, v') \in E\}$. Note that we can view e as an ordinary one-way regular expression with memory over this extended alphabet. A straightforward induction on expressions shows that $(v,v') \in [\![e]\!]^G$, where e is viewed as an two-way query over Σ, if and only if $(v,v') \in [\![e]\!]^{G'}$, where e is now a (one-way)query over Σ'.

The desired upper bounds then follow from query evaluation algorithm in Theorem 4.1.6, since both the alphabet and the graph grow only linearly in size. Note that here we use Lemma 5.1.2 that allows us to switch between graph and path semantics. □

Two-way regular queries with binding (2*RQB*s).

Let Σ be a finite alphabet and $\{x_1, \ldots, x_k\}$ a set of variables. The class of two-way RQBs is defined by the following grammar:

$$e \;:=\; \varepsilon \mid a \mid a^- \mid e+e \mid e\cdot e \mid e^+ \mid e[c] \mid \downarrow\overline{x}.\{e\} \tag{5.2}$$

where a ranges over alphabet letters, c over conditions in $\mathcal{C}_k$, and $\overline{x}$ over tuples of variables from $x_1,\ldots,x_k$.

Graph semantics of 2RQBs is defined with respect to a valuation ν of variables. The *evaluation* $[\![e,\nu]\!]^G$ of an 2RQB e, with respect to a valuation ν over a data graph $G=\langle V,E,\rho\rangle$ is the set of all pairs (v,v') of nodes in V defined recursively as follows:

$$\begin{aligned}
[\![\varepsilon,\nu]\!]^G &= \{(v,v) \mid v\in V\},\\
[\![a,\nu]\!]^G &= \{(v,v') \mid (v,a,v')\in E\},\\
[\![a^-,\nu]\!]^G &= \{(v,v') \mid (v',a,v)\in E\},\\
[\![e_1\cdot e_2,\nu]\!]^G &= [\![e_1,\nu]\!]^G \circ [\![e_2,\nu]\!]^G,\\
[\![e_1\cup e_2,\nu]\!]^G &= [\![e_1,\nu]\!]^G \cup [\![e_2,\nu]\!]^G,\\
[\![e^+,\nu]\!]^G &\quad \text{is the transitive closure of } [\![e,\nu]\!]^G,\\
[\![e[c],\nu]\!]^G &= \{(v,v') \mid (v,v')\in [\![e,\nu]\!]^G, \rho(v'),\nu \models c\},\\
[\![\downarrow x.\{e\}]\!]^G &= \{(v,v') \mid (v,v')\in [\![e,\nu[\overline{x}=\rho(v)]]\!]^G\}.
\end{aligned}$$

Again, that for one-way languages the semantics for 2RQBs extends the one in Section 4.3 is easily shown by induction on the structure of expression.

Proposition 5.1.4. *Let e be any closed regular expression with binding and $Q = x \xrightarrow{e} y$ an RQB. Then for any data graph G it holds that $(v,v')\in [\![e]\!]^G$ if and only if $(v,v')\in Q(G)$.*

Just as with 2RQMs and regular expressions with memory we can also show that graph semantics of *RQB*s can be used to define data path languages in an equivalent way as done for regular expressions with binding in Section 4.3.

Lemma 5.1.5. *For any regular expression with binding e, valuation ν, and any data path w we have that $w\in L(e,\nu)$ if and only if $(v,v')\in [\![e,\nu]\!]^{G_w}$, with G_w as in Figure 5.2.*

Algorithm for solving the query evaluation problem for 2*RQB*s is identical to the one for 2*RQM*s. Thus we obtain the following.

Proposition 5.1.6. *Combined complexity of evaluating 2RQB queries is* PSPACE*-complete. Data complexity if* NLOGSPACE*-complete.*

Two-way regular queries with data tests (2*RQD*s).

The class of two-way RQDs is defined by the following grammar:

$$e \;:=\; \varepsilon \mid a \mid a^- \mid e\cup e \mid e\cdot e \mid e^+ \mid e_= \mid e_{\neq} \tag{5.3}$$

where a ranges over labels from the alphabet Σ.

Note that here we do not consider constant tests given by simplified conditions. It is however readily observed that these can be added without affecting any of the results below.

Graph semantics of 2RQDs is defined in a much simpler way than for RQMs. The *evaluation* $[\![e]\!]^G$ of an 2RQD e over a data graph $G = \langle V, E, \rho \rangle$ is the set of all pairs (v_1, v_2) of nodes in V defined recursively as follows:

$$
\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\},\\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\},\\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\},\\
[\![e_1 \cdot e_2]\!]^G &= [\![e_1]\!]^G \circ [\![e_2]\!]^G,\\
[\![e_1 \cup e_2]\!]^G &= [\![e_1]\!]^G \cup [\![e_2]\!]^G,\\
[\![e^+]\!]^G &\quad \text{is the transitive closure of } [\![e]\!]^G,\\
[\![e_=]\!]^G &= \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) = \rho(v')\},\\
[\![e_\neq]\!]^G &= \{(v,v') \mid (v,v') \in [\![e]\!]^G, \rho(v) \neq \rho(v')\}.
\end{aligned}
$$

As before, one can check that using this semantics restricted to one-way queries yields the same result as when applying semantics from Section 4.4. Namely we have the following.

Proposition 5.1.7. *Let e be any regular expression with equality and $Q = x \xrightarrow{e} y$ an RQD. Then for any data graph G it holds that $(v,v') \in [\![e]\!]^G$ if and only if $(v,v') \in Q(G)$.*

Again, we can use graph semantics of RQDs to define languages of data paths accepted by regular expression with equality e by asserting that a path w is accepted by e if and only if $(v,v') \in [\![e]\!]^{G_w}$, with v, v' and G_w as in Figure 5.2. That this definition is equivalent to the one from Section 4.4 follows from the next lemma, easily shown by induction on e.

Lemma 5.1.8. *For any regular expression with equality e and any data path w we have that $w \in L(e)$ if and only if $(v,v') \in [\![e]\!]^{G_w}$, with G_w as in Figure 5.2.*

Using the same trick of doubling the alphabet with inverse symbols and subsequently using the algorithm from Theorem 4.4.8 we can see that adding inverses has no impact on the computational complexity of query evaluation.

Proposition 5.1.9. *Combined complexity of evaluating 2RQD queries is in* PTIME. *Data complexity is* NLOGSPACE*-complete.*

5.2 Conjunctive queries

A standard extension of RPQs is that to *conjunctive RPQs*, or CRPQs [Calvanese et al., 2000, Deutsch and Tannen, 2001, Florescu et al., 1998]. These add conjunctions of RPQs and existential quantification over variables, in the same way as conjunctive queries extend atomic formulae of relational calculus. We now look at similar extensions of RPQs with data.

Formally, they are defined as expression of the form

$$Ans(\overline{z}) \;:=\; \bigwedge_{1\leq i\leq m} x_i \xrightarrow{L_i} y_i, \tag{5.4}$$

where $m > 0$, each $x_i \xrightarrow{L_i} y_i$ is a query in one of the formalisms from Chapter 4, and $\overline{z}$ is a tuple of variables among $\overline{x}$ and $\overline{y}$. A query with the head $Ans()$ (i.e., no variables in the output) is called a *Boolean* query. To establish terminology we will talk about:

- Conjunctive regular data path queries (CRDPQs), when each $x_i \xrightarrow{L_i} y_i$ is a RDPQ,
- Conjunctive regular queries with memory (CRQMs), when each $x_i \xrightarrow{L_i} y_i$ is an RQM,
- Conjunctive regular queries with binding (CRQBs), when each $x_i \xrightarrow{L_i} y_i$ is an RQB,
- Conjunctive regular queries with data tests (CRQDs), when each $x_i \xrightarrow{L_i} y_i$ is an RQD,
- Conjunctive regular queries with variables (CRQVs), when each $x_i \xrightarrow{L_i} y_i$ is an RQV.

We will also use the name **conjunctive data path query**(CDPQ) for a query from any of the five classes just defined.

These queries extend their base atoms with conjunction, as well as existential quantification: variables that appear in the body but not in the head (i.e., variables in $\overline{x}$ and $\overline{y}$ but not $\overline{z}$) are assumed to be existentially quantified.

The semantics of a CDPQ Q of the form (5.4) over a data graph $G = \langle V, E, \rho\rangle$ is defined as follows. Given a valuation $\nu : \bigcup_{1\leq i\leq m}\{x_i, y_i\} \to V$, we write $(G,\nu) \models Q$ if $(\nu(x_i), \nu(y_i))$ is in the answer of $x_i \xrightarrow{L_i} y_i$ on G, for each $i = 1,\ldots,m$. Then $Q(G)$ is defined as the set of all tuples $\nu(\overline{z})$ such that $(G,\nu) \models Q$. If Q is Boolean, we let $Q(G)$ be true if $(G,\nu) \models Q$ for some ν (that is, as usual, the empty tuple models the Boolean constant true, and the empty set models the Boolean constant false).

As before, we study data and combined complexity of the query evaluation problem, i.e. checking, for a CDPQ Q, a data graph G and a tuple of nodes $\overline{v}$, whether $\overline{v} \in Q(G)$ (for data complexity the query Q is fixed).

First, we show that for all the formalisms studied in the previous chapter, no cost is incurred by going from a single query to a conjunctive query as far as data complexity is concerned.

Theorem 5.2.1. *Data complexity of conjunctive data path queries remains* NL*-complete if they are defined using RDPQs,RQMs, RQBs, RQDs, or RQVs.*

Proof. Consider a query of the form (5.4) and let $\overline{z'}$ be the tuple of variables from $\overline{x}$ and $\overline{y}$ that is not present in $\overline{z}$. To check whether $\overline{v} \in Q(G)$, we need to check whether there exists a valuation $\overline{v'}$ for $\overline{z'}$ so that under that valuation each of the queries in the conjunction in (5.4) is true.

We know from the previous sections that checking whether $v \xrightarrow{L} v'$ evaluates to true for some nodes v, v' can be done with NL data complexity for all the formalisms mentioned in the

theorem. Thus, given a data graph $G = \langle V, E, \rho \rangle$, we can enumerate all the tuples from $V^{|\overline{z'}|}$, and for each of them check the truth of all the queries in conjunction (5.4). Since we deal with data complexity, $|\overline{z'}|$ is fixed, and thus such an enumeration can be done in logarithmic space, showing that query evaluation remains in NL. Note that the NL algorithms can be composed here since they are independent one of another. □

For combined complexity, we have the same bounds for CRDPQs, CRQMs, CRQBs and CRQVs. For CRQDs we get NP-completeness, which matches the combined complexity of conjunctive queries and CRPQs.

Theorem 5.2.2. *Combined complexity of conjunctive regular data path queries remains* PSPACE*-complete if they are specified using RDPQs, RQMs and RQBs. It is* NP*-complete if they are specified using RQDs or RQVs.*

Proof. PSPACE-hardness follows from the corresponding results for *RQB*s, and NP-hardness follows from NP-hardness of relational conjunctive queries. Thus we show upper bounds. The algorithm (using notations from the proof of Theorem 5.2.1) is the same in all the cases: guess a tuple $\overline{v'}$ of nodes for $\overline{z'}$, and check whether all the queries in conjunction (5.4) are true. We know that for RDPQs, RQMs and RQBs the latter can be done in PSPACE; since PSPACE is closed under nondeterministic guesses we have the PSPACE upper bound for combined complexity. For CRQDs, an NP upper bound for the algorithm follows from the PTIME bound for combined complexity for RQDs. Finally, for CRQVs we will also guess a path between the nodes corresponding to x_i, y_i (along with an associated witnessing pattern), which are by Theorem 4.5.6 of polynomial size. We can then verify our guess in PTIME, thus obtaining the desired bound. □

All of the complexity bounds for languages considered in this section are summarized in the following table.

Query answering	*CRDPQ*	*CRQM*	*CRQB*	*CRQD*	*CRQV*
data complexity	NL-complete	NL-complete	NL-complete	NL-complete	NL-complete
combined complexity	PSPACE-complete	PSPACE-complete	PSPACE-complete	NP-complete	NP-complete

Table 5.1: Summary of complexity bounds for classes of conjunctive queries

5.3 Adding variables to register automata

In the previous chapter we proved that variable automata are incomparable in terms of expressive power with register automata and regular expressions with binding. In particular we showed that they can express a property that all data values differ from the last, a feature know

not to be expressible by register automata. On the other hand, bound variables in variable automata behave like a limited version of registers that are capable of storing a data value only once. As the result, variable automata are not able to express even some simple properties definable by regular expressions with equality.

In this section we define a general model that will encompass both register and variable automata and study its query evaluation problem over graphs. The model is essentially a variable automaton that can use the full power of registers in a same way that an ordinary register automaton would. Another way to look at it is as adding the free variable and constants to register automata. It will subsume both models, but we shall see that it does not increase the complexity of query evaluation beyond PSPACE.

Definition 5.3.1. *Let Σ be a finite alphabet, k a natural number and C a finite set of data values. A* k-register automaton with variables *(or varRA for short) is a tuple $\mathcal{A} = (Q, q_0, F, \delta, \tau_0, \{\star\}, C)$, where:*

- *$Q = Q_w \cup Q_d$, where Q_w and Q_d are two finite disjoint sets of word states and data states;*
- *$q_0 \in Q_d$ is the initial state;*
- *$F \subseteq Q_w$ is the set of final states;*
- *$\tau_0 \in \mathcal{D}^k$ is the initial configuration of the registers;*
- *$\delta = (\delta_w, \delta_d)$ is a pair of transition relations:*
  - *$\delta_w \subseteq Q_w \times \Sigma \times Q_d$ is the word transition relation;*
  - *$\delta_w \subseteq Q_d \times C_k \times 2^{[k]} \times Q_w \quad \bigcup \quad Q_d \times \{C \cup \{\star\}\} Q_w$ is the data transition relation.*

Note that the data transition relation has three different types of transitions. The first type is of the form (q, c, I, q') and is the same as in Definition 6.1.1. The second type checks if a given data value is a constants and is of the form (q, d, q') with $d \in C$. Finally, the last type is of the form $(q, \star, q')$ and we will refer to such transitions as $\star$-transitions.

We now define the notion of acceptance. A k-register automaton $\mathcal{A}$ with variables accepts a data path $w = d_0 a_0 d_1 a_1 \ldots a_{n-1} d_n$ if there is a sequence $q_0, q'_0, q_1, q'_1, \ldots, q_n, q'_n$ of states in Q with $q'_n \in F$, a sequence $t_0, t'_0, \ldots t_{n-1}, t'_{n-1}, t_n$ of transitions and a sequence $\tau_1, \ldots \tau_n$ of register assignments such that:

- For $i = 1 \ldots n$ we have $t'_i = (q'_i, a, q_{i+1})$ and $a_i = a$;
- For $i = 0 \ldots n$ each t_i is a data transition and precisely one of the following holds:
  1. If $t_i = (q_i, c, I, q'_i)$, then $\tau_i, d_i \models c$ and τ_{i+1} is obtained by storing d_i in registers from I ;
  2. If $t_i = (q_i, d, q'_i)$, then $d_i = d$;

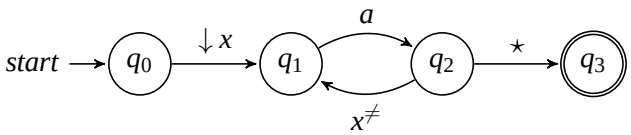
3. If $t_i = (q_i, \star, q_i')$, then $d_i = d_j$ iff $t_j = (q_j, \star, q_j')$.

Register automata with variables can use standard register automata transitions, as well as check if some data value matches a constant. Additionally, by allowing $\star$-transitions, they can state that some value will not be stored in the registers. Note that, unlike standard automata transitions, $\star$-transitions are global in character – that is, they do not refer only to the next and the previous state in a run, but to the run as a whole.

It is apparent that register automata with variables extend both register and variable automata in a natural way. Moreover, if we restrict the registers by allowing them to store values only once and restrict conditions to single equality tests only, we get variable automata. On the other hand if we disallow the usage of the free variable $\star$ we get register automata.

In the previous Chapter we have seen several examples of properties expressible by register automata and variable automata. Next we show that with varRA we can define data path languages not expressible by either of them.

Example 5.3.2. *The language of all data paths where both the first and the last data value differ from all other data values is defined by the following varRA.*



Here the first three states make sure that first data value is not equal to any value before the last. Finally the $\star$-transition taking us to the final state makes sure that no other value is equal to it. Note that this automaton depends on the fact that $\star$-transitions can reason about complete runs of an automaton and not just adjacent transitions.

We can now define a class of graph queries based on register automata with variables in the same way as we did for other data path formalisms in Chapter 4. We will call such queries register queries with variables.

Definition 5.3.3. *A* register query with variables (RQVar) *is an expression $Q = x \xrightarrow{\mathcal{A}} y$ where $\mathcal{A}$ is a register automaton with variables.*

Given a data graph G, the result of the query $Q(G)$ consists of pairs of nodes (v, v') such that there is a data path w from v to v' that belongs to $L(\mathcal{A})$.

Surprisingly, despite the increased expressive power, this model still retains the complexity of register automata.

Theorem 5.3.4.
- *Combined complexity of RQVar queries is* PSPACE*-complete.*
- *Data complexity of RQVar queries is* NL*-complete.*

Proof. To prove this we use a similar construction to the one used in the proof of Theorem 4.1.6. We start by showing that, given a finite set of data values D and a k-register automaton with variables $\mathcal{A}$, we can produce a variable automaton $\mathcal{A}_D$ that accepts precisely the same words as $\mathcal{A}$ does when both use only data values from D.

Let $\mathcal{A} = (Q, q_0, F, \delta, \tau_0, \{\star\}, C)$ be a k-register automaton with variables and D a finite set of data values.

We define the desired VFA $\mathcal{A}_D = (Q', q_0', F', \Gamma, \delta')$ as follows:

- $\Gamma = \{C \cup D\} \cup \{\star\}$
- $Q' = Q \times D_0^k$, where $\perp$ is a new data value not in D and $D_0 = D \cup \{\perp\} \cup \{\tau_0(i) | i = 1 \dots k\}$
- $q_0' = (q_0, \tau_0)$
- $F' = F \times D_0^k$
- For the transitions:
  - If $(q, a, q') \in \delta_w$ we add
    $$((q, \tau), a, (q', \tau))$$
    to δ_w', for every assignment τ
  - If $(q, c, I, q') \in \delta_d$, we add
    $$((q, \tau), d, (q', \tau'))$$
    to δ_d', for every data value $d \in D$ and assignments τ, τ' such that $\tau, d \models c$ and τ' is obtained by storing d into registers from I
  - If $(q, d, q') \in \delta_d$, with d a constant in C we add
    $$((q, \tau), d, (q', \tau))$$
    to δ_d', for every assignment τ
  - If $(q, \star, q') \in \delta_d$ we add
    $$((q, \tau), \star, (q', \tau))$$
    to δ_d', for every assignment τ.

Note that our VFA $\mathcal{A}_D$ uses no bound variables.

Next we prove that the variable automaton obtained in this construction indeed accepts the same class of data paths over D as the original register automaton with variables does.

Claim 5.3.5. *Let w be a data path whose data values come from D. Then $w \in L(\mathcal{A}_D)$ if and only if $w \in L(\mathcal{A})$.*

Proof. Assume first that $w = d_0 a_0 \dots a_{n-1} d_n$, where $d_0, \dots d_n$ are from D, is accepted by $\mathcal{A}_D$.

Since $\mathcal{A}_D$ is a VFA with constants and free variable only (and no bound variables), this means that there is a witnessing pattern $v = v_0 b_0 \dots b_{n-1} v_n$ and a sequence $(q_0, \tau_0), (q'_0, \tau'_0), \dots, (q_n, \tau_n), (q'_n, \tau'_n)$ of states in $\mathcal{A}_D$, with $(q'_n, \tau'_n) \in F'$ such that:

1. for each i we have $(q_i, v_i, q'_i) \in \delta_d$ and $(q'_i, b_i, q_{i+1}) \in \delta_w$,
2. $a_i = b_i$ and $(q'_i, a_i, q_{i+1}) \in \delta_w$, for $i = 0, \dots, n$,
3. if $v_i = d \in C$ then $(q_i, d, q'_i) \in \delta_d$ and $d_i = d$,
4. if $v_i = \star$ and $v_j \neq \star$ then $d_i \neq d_j$.

But then this sequence of states and transitions of $\mathcal{A}_D$ can be easily transformed into an accepting run of $\mathcal{A}$ on w (follows from the construction of $\mathcal{A}_D$), thus implying that $w \in L(\mathcal{A})$.

To see that the reverse is true we simply transform the accepting run of $\mathcal{A}$ on w into the matching run of $\mathcal{A}_D$. The witnessing pattern for w will be obtained by converting every data value matched with $\star$ in w by $\star$ itself. All the details easily follow from the definition of acceptance and the construction of $\mathcal{A}_D$. □

To complete the proof of Theorem 5.3.4 we use the same technique as in the proof of Theorem 4.5.7.

As input we are given a query Q, specified by a register automaton with variables $\mathcal{A}$ and a data graph G, together with two nodes s and t. Let $D = \mathcal{D}(G)$ be the set of all data values appearing in G.

We again view our graph as a VFA (with the initial state s_s and final state t_t) and denote it by $\mathcal{A}_G$. We can now build the product of $\mathcal{A}_G$ and $\mathcal{A}_D$. Testing his automaton for nonemptiness is the same as answering our query evaluation problem.

Note that the number n_1 of states of $\mathcal{A}_D$ is $O(|\mathcal{A}| \times |D|^k)$, the number of bound variables $d_1 = 0$ and the number of constants c_1 at most $|D| + |\mathcal{A}|$.

For $\mathcal{A}_G$ we have $n_2 = O(|G|)$, while $d_2 = 0$ and $c_2 = |D|$.

By the construction in [Grumberg et al., 2010b] we know that the size of the product is $O(n_1 \cdot n_2 \cdot \frac{(c_1+c_2+d_1+d_2)!}{(c_1+c_2)!}) = O(n_1 \cdot n_2)$.

Using the values above we get that the size is $O(|\mathcal{A}| \times |D|^k \times |G|)$.

Since $|D| = |G|$ this is polynomial in $|G|$ if the automaton is fixed and exponential if it is part of the input (as the number of registers gets into the exponent). Thus using the standard on-the-fly method for testing nonemptiness we obtain the desired result. □

Despite their high expressive power and acceptable evaluation bounds, it is highly unlikely that regular queries with variables might be of interest to graph database practitioners due to their added complexity. Indeed, to specify a query in this formalism requires a lot of care and

even simple queries are quite cumbersome to write. Thus, despite good algorithmic properties and a wide variety of queries they can express, we will not try to promote RQVs as a querying standard for data graphs (as far as path queries are concerned), since a language suited for that role should strike a fine balance between expressive power, efficiency and ease of use.

Chapter 6

The language theory gap

In Chapter 4 we developed several classes of queries for data graphs. As we have seen all of these classes were based on an underlying automaton model, or a class of expressions defining data paths. Therefore formalisms used to define path queries have an intrinsically language theoretic flavour and there are many interesting questions about them that fall out of scope when approached from a purely database theoretic point of view. Indeed, register automata, for example, were originally introduced to describe languages over infinite alphabets [Kaminski and Francez, 1994], and later extended to operate over data words [Demri and Lazić, 2009, Segoufin, 2006], a setting that, as we have already discussed in Chapter 3, is very close to that of data paths.

Such setting, where languages draw their letters not only from a finite alphabet, as is the case with NFAs or context-free grammars, but also from an infinite set of data objects, has received a lot of attention recently due to applications in program verification and XML. In particular, data word languages are commonly used to model infinite state systems [Demri and Lazić, 2009, Segoufin, 2006, Bouajjani et al., 2003] and to reason about static properties of XML documents [Figueira, 2010b, Segoufin, 2007, Neven et al., 2004]. In these scenarios questions like nonemptiness and membership naturally come into play as they relate to checking if a class of documents or programs respects some structural property. Furthermore, another common language theoretic question, that of language containment and universality, is naturally linked to program or query equivalence, an issue particularly important when doing optimisation.

All of this warrants a language theoretic study of data path formalisms we introduced in Chapter 4 and that is what we do in the present chapter. As just mentioned, here it is more interesting to define our formalisms over data words, however, as already discussed, these two approaches are equivalent. For this reason we will redefine all of the formalisms from Chapter 4 to specify data words instead of data paths, while still keeping the original terminology in order to reduce proliferation of different names for same classes of expressions or automata.

Therefore we will still be working with e.g. regular expressions with memory, the only difference being that these will now specify data words and not data paths. In what follows we will also remove constant tests from our expressions and automata, as these are seldom used in language theory, although all of the results still hold if they are present. This is mainly done for the ease of notation and to make our presentation more precise.

We begin with the study of register automata. Note that questions such as nonemptiness, membership, universality and the important closure properties were already considered in e.g. [Sakamoto and Ikeda, 2000, Neven et al., 2004, Kaminski and Francez, 1994]. However, it was observed in [Demri and Lazić, 2009] that subtle changes to the model can lead to different complexity bounds for some of these problems. For example, allowing the automata to have the same data value stored in more then one register and allowing explicit inequality comparisons makes them more intuitive, but it also increases the complexity of nonemptiness [Demri and Lazić, 2009]. The model of register automata used here is essentially equivalent to the one in [Demri and Lazić, 2009], however the notation is different, so in line with the previous remarks about slight changes affecting some of the complexity bounds, we will reprove all of the results to have a self contained study.

Following this, we will see how to modify the definition of the three classes of expressions introduced in Chapter 4 and study their closure properties and standard decision problems. In the end we also expand the definition of variable automata from [Grumberg et al., 2010a], where they were used to define words over an infinite alphabet, to the setting of data words, showing that all of the results still hold here.

Basic definitions We will now shortly recall the definition of data words and formally define standard decision problems and closure properties that we study in the following sections.

A **data word** is simply a finite string over the alphabet $\Sigma \times \mathcal{D}$, where Σ is a finite set of letters and $\mathcal{D}$ an infinite set of data values. That is, in each position a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will denote data words by $\binom{a_1}{d_1}\ldots\binom{a_n}{d_n}$, where $a_i \in \Sigma$ and $d_i \in \mathcal{D}$. An example of a data word over the alphabet $\Sigma = \{a,b,c\}$ and the set $\mathbb{N}$ of integers as data values is:

$$\binom{a}{3}\binom{a}{7}\binom{c}{1}\binom{b}{3}\binom{a}{1}.$$

The set of all data words over the alphabet Σ and the set of data values $\mathcal{D}$ is denoted by $(\Sigma \times \mathcal{D})^*$. A data word language is simply a subset $L \subseteq (\Sigma \times \mathcal{D})^*$.

Standard decision problems Some of the most important standard decision problems in formal language theory are membership, nonemptiness, language containment and universality. In this chapter we will examine all of these problems for each of the formalisms we introduce and determine whether they are decidable, and if they are, what is their computational complexity.

Next we define the problems formally.

Let $\mathcal{C}$ be a class of automata, or expressions, defining languages of data words over some fixed finite alphabet Σ. The nonemptiness problem asks, given an automaton, or an expression over the alphabet Σ, are there any data words in the language of this expression or automaton. Formally we have:

NONEMPTINESS($\mathcal{C}$)	
Input:	An expression, or an automaton $\mathcal{A} \in \mathcal{C}$.
Task:	Decide whether $L(\mathcal{A}) \neq \emptyset$.

When considering data word formalisms in this chapter we will also examine the complexity of the membership problem, that is the problem of checking, for an expression (or an automaton) and a data word, if this word belongs to the language of the automaton. The membership problem is defined as follows.

MEMBERSHIP($\mathcal{C}$)	
Input:	An expressions, or an automaton $\mathcal{A} \in \mathcal{C}$ and a data word $w \in (\Sigma \times \mathcal{D})^*$.
Task:	Decide whether $w \in L(\mathcal{A})$.

Another problem we will consider when studying properties of formalisms defining data word languages is language universality. Here we will ask, given an expression (or an automaton) over some fixed finite alphabet Σ, whether it generates all the words from $(\Sigma \times \mathcal{D})^*$. The language universality problem is defined below.

UNIVERSALITY($\mathcal{C}$)	
Input:	An expression, or an automaton $\mathcal{A} \in \mathcal{C}$ over Σ and $\mathcal{D}$.
Task:	Decide whether $L(\mathcal{A}) = (\Sigma \times \mathcal{D})^*$.

An important generalisation of universality is the language containment problem. Here we simply ask, given two expressions or automata, if every data word in the language of the first one is also contained in the language of the second one. Given the close connection of path queries and language theoretic formalisms used to define them, it comes as a no surprise that this problem is basically equivalent to query containment, an issue which we will address in Chapter 10. Next we define language containment problem formally.

CONTAINMENT($\mathcal{C}$)	
Input:	Two expressions, or automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in $\mathcal{C}$.
Task:	Decide whether $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$.

Closure properties Another important class of questions regarding language defining formalisms are closure properties. Indeed, it is crucial to determine if a language defining formalism is closed under certain properties to be able to build more complex languages starting from simpler ones. Some of the most commonly studied closure properties are:

1. *Union*, which asks, given two languages definable by some formalism, if their union is also definable.
2. *Intersection*, asking if the intersection of two languages is definable in some formalism if the languages themselves are.
3. *Complement*, asking if one can define the set theoretic complement of a given language.
4. *Concatenation*, asking if concatenation of two definable languages is also definable.
5. *Kleene star*, determining if the language containing arbitrary long iterations of a word from the starting language is definable.

In this chapter we will examine closure properties of each of the proposed formalisms for defining data word languages. While all of these properties are important, exclusion of some of them does not necessarily render a language unusable. Indeed, while the class of regular languages is known to be closed under all of the above mentioned properties, context free languages lack closure under intersection and complementation [Hopcroft and Ullman, 1979], but are still heavily used in compiler design, programming languages and pattern matching. Similar behaviour will be witnessed by the languages we study in this chapter. In particular, none of the languages will be closed under complementation, as already discussed in Section 3.2, while some will be shown not to be closed under intersection either.

6.1 Register automata

Register automata are an analogue of NFAs for data words. They move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to ones previously stored into the registers. Our version of register automata will use conditions which are boolean combinations of atomic $=, \neq$ comparisons of data values.

Conditions are defined in the same manner as in Section 4.1. For the sake of readability we define them here again adding some additional syntactic sugar to ease the notation. To define conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then the set of conditions C_k is given by the grammar:

$$c \;:=\; \mathtt{tt} \mid \mathtt{ff} \mid x_i^{=} \mid x_i^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \quad 1 \leq i \leq k.$$

As before, the satisfaction is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}^k$ as follows:

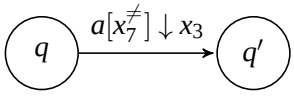
- $d,\tau \models \mathtt{tt}$ and $d,\tau \not\models \mathtt{ff}$;
- $d,\tau \models x_i^{=}$ iff $d = d_i$;
- $d,\tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d,\tau \models c_1 \wedge c_2$ iff $d,\tau \models c_1$ and $d,\tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d,\tau \models \neg c$ iff $d,\tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1,\ldots,k\}$.

Definition 6.1.1 (Register data word automata)**.** *Let Σ be a finite alphabet and k a natural number. A k-*register data word automaton*, or* RA *for short, is a tuple $\mathcal{A} = (Q, q_0, F, T)$, where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$F \subseteq Q$ is the set of final states;*
- *T is a finite set of transitions of the form $(q,a,c) \to (I,q')$, where q,q' are states, a is a label, $I \subseteq [k]$, and c is a condition in $\mathcal{C}_k$.*

Intuitively the automaton traverses a data word from left to right, starting in q_0, with all registers empty. If it reads $\binom{a}{d}$ in state q with register configuration τ, it may apply a transition $(q,a,c) \to (I,q')$ if $d,\tau \models c$; it then enters state q' and changes contents of registers i, with $i \in I$, to d. We will represent register data word automata transitions graphically as follows:



A typical transition in a data word automaton.

Here we assume that the value is compared to the one stored in the register corresponding to x_7 and later on stored into the one corresponding to x_3.

To define acceptance formally we first define a configuration of a k-register data word automaton $\mathcal{A}$ on data word $w = \binom{a_1}{d_1}\ldots\binom{a_n}{d_n}$ as a triple (q,j,τ), where q is the current state of $\mathcal{A}$, j is the current position of the symbol in w that $\mathcal{A}$ reads and τ is the current state of the registers. We use the symbol $\perp$ to indicate that a register is unassigned; that is, τ is a k-tuple over $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$. The initial configuration is $(q_0,1,\tau_0)$, where $\tau_0 = (\perp,\ldots,\perp)$, and any configuration (q,j,τ) with $q \in F$ is a final configuration.

From a configuration (q,j,τ) we can move to a configuration $(q',j+1,\tau')$ if:

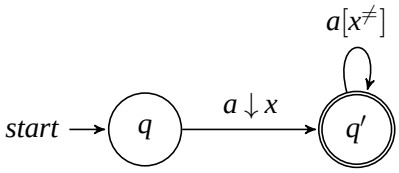
- $(q,a_j,c) \to (I,q')$ is a transition in $\mathcal{A}$,
- $d_j,\tau \models c$ and
- τ' is obtained from τ by replacing data values in registers from I by d_j.

We say that $\mathcal{A}$ accepts w if there is a sequence of configurations of $\mathcal{A}$ on w that leads $\mathcal{A}$ from the initial to a final configuration while reading w.

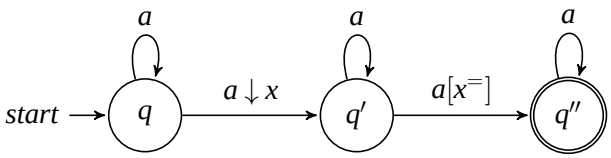
Remark Given a k-register data word automaton $\mathcal{A}$ and a tuple $\tau \in \mathcal{D}_{\perp}^{k}$, we can turn $\mathcal{A}$ into an automaton $\mathcal{A}(\tau)$ defined just as $\mathcal{A}$ but starting with τ as the register configuration. Such an extension does not affect the class of accepted languages, but will be useful in inductive constructions when automata need not start with all registers unassigned.

Example 6.1.2. *Next we present two examples of register automata and languages they define.*

- *The data word language where all data values are different from the first (and the label is a^*) is defined by the following register automaton:*



- *The language of data words having two equal data values (and where the label is a^*) is given by the following automaton:*



Language theoretic properties

In this section we recall the basic language theoretic properties of register data word automata. Most of these results follow from [Kaminski and Francez, 1994], however, since some subtle differences were introduced to the model we will reprove most of the results to make the presentation self contained. Some changes introduced here will have an impact on the nonemptiness problem, as already noted in [Sakamoto and Ikeda, 2000, Demri and Lazić, 2009], however, all of the other results remain intact. In order to prove complexity results about membership and nonemptiness we will require some general properties of register automata that we examine next. At the end we will also recall closure properties of the class of languages defined by register automata.

General properties of register automata A useful property of register automata that will be needed in what follows is that, intuitively, such automata can only keep track of as many data values as can be stored in their registers. Formally, we have:

Lemma 6.1.3. *Let $\mathcal{A}$ be a k-register data word automaton. If $\mathcal{A}$ recognizes some word of length n, then it recognizes a word of length n that uses at most $k+1$ different data values.*

Proof. We first set some notation. We will say that two k-register assignments τ and $\overline{\tau}$ are of the same equality type if we have $\tau(i) = \tau(j)$ if and only if $\overline{\tau}(i) = \overline{\tau}(j)$, for all $i, j \leq k$. Note that this also implies that $\tau(i) \neq \tau(j)$ if and only if $\overline{\tau}(i) \neq \overline{\tau}(j)$.

We will prove a slightly more general claim, allowing our automata to start with an nonempty assignment of the registers. Let $\mathcal{A}(\tau_0) = (Q, q_0, F, T)$ be a k-register data word automaton, starting with the initial assignment τ_0 in the registers and $w = \binom{a_1}{d_1} \dots \binom{a_n}{d_n}$ a word that it accepts. This means that there is a sequence of states $q_0, q_1, \dots, q_n$, with $q_n \in F$ and a sequence of register assignments $\tau_0, \tau_1, \dots, \tau_n$ such that $(q_{i-1}, a_i, c_i) \to (I_i, q_i) \in T$, that $\tau_{i-1}, d_i \models c_i$ and τ_i is obtained from $\tau_{i-1}$by replacing all registers from I_i with d_i, for $i = 1 \dots n$.

Now let $\overline{S} = \{\tau_0(i) : 1 \leq i \leq k\} - \{\perp\}$. That is $\overline{S}$ contains all the data values from the initial assignment, except the one denoting that the register is empty.

Let S be any set of data values such that $|S| = k+1$ and $\overline{S} \subseteq S$.

We prove by induction on $i \leq n$ that we can define a data word w_i, of length i, such that $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$, where $a_1, \dots a_i$ are from w and $d_1^i, \dots, d_i^i$ are from S. We then show that for this w_i there is a sequence of assignments $\tau'_0, \tau'_1, \dots \tau'_i$ such that each τ'_j is of the same equality type as τ_j, where $j \leq i$ and it holds that $\tau_{j-1}, d_j \models c_j$, for all $j \leq i$ and each τ'_j is obtained from τ'_{j-1} by replacing all the data values from I_j by d_j. Note that this actually means that $\mathcal{A}$ goes through the same sequence of states while reading w_i as it did while reading w. But then w_n is the desired word from the statement of the lemma.

To prove this we first assume that $i = 1$. We set $\tau'_0 = \tau_0$ and select $d \in S$ such that $\tau_0, d \models c_1$ (note that this is possible since we have $k+1$ values at disposal and test only for equality or inequality with a fixed set of k elements) and such that τ_1 and τ'_1 are of the same equality type, where τ'_1 is obtained from τ'_0 by replacing all data values from I_1 by d. Again, this is possible since the original d_1 (from w) could have either been different from all data values in τ_0 or equal to some of them, a choice we can simulate with elements from S. We now set $w_1 = \binom{a_1}{d}$.

Assume now that the claim holds for $i < n$. We prove the claim for $i+1$. By the induction hypothesis we know that there exists a data word $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$ with data values from S and a sequence of assignments each one obtained from the previous by the condition dictated by the original accepting run that allow $\mathcal{A}$ to go through the states $q_0, q_1, \dots, q_i$. We now pick $d \in S$ such that $\tau'_i, d \models c_{i+1}$ and τ'_{i+1}, obtained from τ'_i by replacing all data values from I_{i+1} by d, has the same equality type as τ_{i+1}. Note that this is possible since τ_i and τ'_i have the same equality type by the induction hypothesis and we have enough data values at our disposal (again, we have to pick d so that it is in the same relation to data values from τ'_i as d_{i+1} from w was to data values from τ_i, but this is possible since each assignment can remember at most k data values). Now we simply define $w_{i+1} = w_i \cdot \binom{a_{i+1}}{d}$. Note that this w_{i+1} has all the desired properties and

can take $\mathcal{A}$ from q_0 to q_{i+1}.

This concludes the proof of the lemma. □

We now show that we can view register automata as NFAs when restricted only to a finite set of data values. Note that this construction follows the same idea as when done for data paths in Section 4.1. For the sake of completeness, and since the notation differs in the two cases, we also include it here.

Let $\mathcal{A} = (Q, q_0, F, T)$ be a k-register data word automaton, D a finite set of data values, and $D_\perp = D \cup \{\perp\}$. We transform $\mathcal{A}$ into an NFA $\mathcal{A}_D = (Q', q_0', F', \delta)$ over the alphabet $\Sigma \times D$ as follows:

- $Q' = Q \times D_\perp^k$;
- $q_0' = (q_0, \perp^k)$;
- $F' = F \times D_\perp^k$;
- Whenever we have a transition $(q, a, c) \to (I, q')$ in T, we add the transition

$$((q,\tau), \begin{pmatrix} a \\ d \end{pmatrix}, (q', \tau'))$$

to T if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.

It is straightforward to check that $\mathcal{A}$ accepts a data word over $\Sigma \times D$ if and only if $\mathcal{A}_D$ does. That is we obtain the following.

Lemma 6.1.4. *Let D be a finite set of data values and $\mathcal{A}$ a register automaton over Σ. Then there exists a finite state automaton $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ such that $w \in L(\mathcal{A}_D)$ iff $w \in L(\mathcal{A})$, for every w with data values from D. Moreover, $\mathcal{A}_D$ is of size exponential in the size of $\mathcal{A}$ and polynomial in the size of D.*

Decision problems Membership, nonemptiness and universality are some of the most important decision problems related to formal languages. We now recall the exact complexity of these problems for register automata. Since the model of register automata we use here differs slightly from the one in previous work, we sketch how these results carry over to our model.

Recall that nonemptiness problem for an automaton $\mathcal{A}$ is checking whether $L(\mathcal{A}) \neq \emptyset$.

Fact 6.1.5 ([Demri and Lazić, 2009])**.** *The nonemptiness problem for register data word automata is* PSPACE*-complete.*

The lower bound will follow from Theorem 6.2.3 and Proposition 6.2.5. For the upper bound we convert our k-register automaton $\mathcal{A}$ into an NFA $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ (as in the Lemma 6.1.4), where $D = \{0, \ldots, k+1\}$. We know that $\mathcal{A}_D$ recognizes all data words from

$L(\mathcal{A})$ using only data values from D. By Lemma 6.1.3 and invariance under automorphisms (see Fact 6.1.9), we know that checking $\mathcal{A}$ for nonemptiness is equivalent to checking $\mathcal{A}_D$ for nonemptiness. Using on-the-fly construction we get the desired result (note that $\mathcal{A}_D$ can not be created before checking it for nonemptiness).

Remark 5. *It is important to note that subtle differences in the definition of the automaton can lead to slightly better complexity bounds. Indeed, the model used in [Sakamoto and Ikeda, 2000] allows each value to be stored in only one register and imposes some further restrictions, thus bringing the complexity of nonemptiness problem down to* NP*-complete. Here we have opted for a more intuitive approach, that has now become commonly used [Demri and Lazić, 2009, Segoufin, 2006].*

The membership problem asks, for an automaton $\mathcal{A}$ and a word w, whether $w \in L(\mathcal{A})$.

Fact 6.1.6 ([Sakamoto and Ikeda, 2000])**.** *The membership problem for register data word automata is* NP*-complete.*

The lower bound will follow from Theorem 6.2.3 and Proposition 6.2.6. For the upper bound it simply suffices to guess an accepting run of the automaton. Since every transition of the automaton processes one symbol of our data word, we only need to guess $|w|$ states of the automaton, where w is the input data word. It is straightforward to check that we can simulate the automaton in PTIME.

On the other hand, universality and containment problems are undecidable.

Fact 6.1.7 ([Kaminski and Francez, 1994])**.** *Both universality and language containment problems for register data word automata are undecidable.*

It turns out that when no inequality comparisons are allowed in the conditions the problem becomes decidable.

Fact 6.1.8 ([Tal, 1999])**.** *Containment and universality problems are decidable for register automata that compare data values for equality only.*

Closure properties Since register automata closely resemble classical finite state automata, it is not surprising that some (although not all) constructions valid for NFAs can be carried over to register automata. We now recall results about closure properties of register automata [Kaminski and Francez, 1994]. Although our notion of automata is slightly different than the one used there, all constructions from [Kaminski and Francez, 1994] can be easily modified to work in the setting proposed here.

Fact 6.1.9 ([Kaminski and Francez, 1994])**.** *1. The set of languages recognized by register automata is closed under union, intersection, concatenation and Kleene star.*

2. *Languages recognized by register automata are not closed under complement.*

3. *Languages recognized by register automata are closed under automorphisms: that is, if $f : \mathcal{D} \to \mathcal{D}$ is an automorphism and w is accepted by $\mathcal{A}$, then the data word $f(w)$ in which every data value d is replaced by $f(d)$ is also accepted by $\mathcal{A}$.*

Closure under union and Kleene star is apparent immediately. To see that the automata are closed under intersection the product construction is used. The usual powerset construction, however, does not yield an automaton defining the complement of a given language as demonstrated in [Kaminski and Francez, 1994].

6.2 Regular expressions with memory

In order to develop an expression analogue for register data path automata in Section 4.2 we introduced regular expressions with memory. These expression, based on the idea of storing data values in variables were defined to work over data paths. Here we show that they can also be used to specify data word languages. In fact, we will see that using the idea of storing data values in variables (and comparing them using conditions) gives rise to a class of expressions capturing register data word automata in the same way as the usual regular expressions capture regular languages. To do this notice that register automata can be pictured as finite state automata whose transitions between states have labels of the form $a[c]\downarrow I$, where I is a set of registers. Such an automaton can move from one state to another using an arrow $a[c]\downarrow I$ if the letter it sees is a, and the data value (together with the current register assignment) satisfies the condition c. It then proceeds to the next state and updates the registers in I with the current data value. This suggests that the basic building blocks for our expressions will be expressions of the form $a[c]\downarrow I$. Note that this is in a way analogous to how ordinary regular expressions are defined based on the fact that NFA transitions have the label a, and move to the next state if this letter can be matched in the word during a run. Similarly as in the case of NFAs and regular expressions, we will define regular expressions with memory starting from register automata edge labels and closing them under union, concatenation and Kleene star.

Definition 6.2.1 (Expressions with memory)**.** *Let Σ be a finite alphabet and $x_1, \ldots, x_k$ a finite set of variables. Regular expressions with memory, or REM for short, over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

- *ε and $\emptyset$ are expressions;*
- *$a[c]\downarrow I$ is an expression; here $a \in \Sigma$, c is a condition in C_k, and $I \subseteq \{x_1, \ldots, x_k\}$;*
- *If e, e_1, e_2 are expressions, then so are $e_1 + e_2$, $e_1 \cdot e_2$, and e^*.*

For convenience we will write just a if $I = \emptyset$ and the condition $c = \mathtt{tt}$ and similarly when only one of them can be ignored. Also, if $I = \{x\}$, we write $a[c]{\downarrow}x$, or $a{\downarrow}x$ when $c = \mathtt{tt}$, instead of $a[c]{\downarrow}I$.

To define the semantics, we first define what it means for an expression e over $\Sigma[x_1, \ldots x_k]$, a data word w and a tuple $\sigma \in \mathcal{D}_\perp^k$ to infer another tuple $\sigma' \in \mathcal{D}_\perp^k$, viewed as partial assignment of values to variables. We do this inductively on e.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = \varepsilon$ and $\sigma' = \sigma$.
- $(a[c]{\downarrow}I, w, \sigma) \vdash \sigma'$ iff $w = \binom{a}{d}$ and $\sigma, d \models c$ and σ' is obtained from σ by assigning d to each $x_i \in I$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^*, w, \sigma) \vdash \sigma'$ iff
  1. $w = \varepsilon$ and $\sigma = \sigma'$, or
  2. $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e, w_1, \sigma) \vdash \sigma''$ and $(e^*, w_2, \sigma'') \vdash \sigma'$.

We say that a regular expression e *induces* a tuple $\sigma \in \mathcal{D}_\perp^k$ on a data word w if $(e, w, \perp^k) \vdash \sigma$. We then define $L(e)$, the language of e, as the set of all data words on which e induces some tuple σ. A regular expression with memory e is *well-formed* if every variable is bound before being used in a condition. From now on we will assume that all our expressions are well-formed.

Example 6.2.2. *We now give a few examples of data word languages definable by regular expressions with memory.*

1. *The expression $(a{\downarrow}x) \cdot (b[x^{\neq}])^*$ defines the language of data words where word part reads ab^* and such that the first data value is different from all others. It binds while reading the first a, and then it proceeds checking that the letter is b and condition $x^{\neq}$ is satisfied, which is expressed by $b[x^{\neq}]$; the expression is then put in the scope of * to indicate that the number of such values is arbitrary.*
2. *The language of data words in which two data values are the same is given by the expression $\Sigma^* \cdot (\Sigma{\downarrow}x) \cdot \Sigma^* \cdot (\Sigma[x^{=}]) \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and $\Sigma{\downarrow}x$ is a shorthand for $a_1{\downarrow}x + \ldots + a_l{\downarrow}x$. It says: at some point, bind x, and then check that after one or more letters, we have the same data value.*

3. *The language of data words in which the last two data values occur elsewhere in the word with label a is defined by* $\Sigma^* \cdot (a{\downarrow}x) \cdot \Sigma^* \cdot (a{\downarrow}y) \cdot \Sigma^* \cdot (\Sigma[x^=] + \Sigma[y^=]) \cdot (\Sigma[x^=] + \Sigma[y^=])$.

Equivalence with register automata

In this section we prove that every language recognized by register automata can also be described by a regular expression with memory and vice versa. In fact, we show a tighter connection, from which the equivalence will follow. Let $L(e,\sigma,\sigma')$ be the set of all data words w such that $(e,w,\sigma) \vdash \sigma'$, and let $L(\mathcal{A},\sigma,\sigma')$ be the set of all data words w such that w is accepted by $\mathcal{A}(\sigma)$, and there exists an accepting run that ends with a register configuration σ'.

Theorem 6.2.3. 1. *For every regular expression with memory e over* $\Sigma[x_1,\ldots,x_k]$ *there exists (and can be constructed in logarithmic space) a k-register data word automaton* $\mathcal{A}_e$ *such that* $L(e,\sigma,\sigma') = L(\mathcal{A}_e,\sigma,\sigma')$ *for every* $\sigma,\sigma' \in \mathcal{D}_\perp^k$.

2. *For every k-register data word automaton* $\mathcal{A}$ *there exists (and can be constructed in exponential time) a regular expression with memory* $e_\mathcal{A}$ *over* $x_1,\ldots,x_k$ *such that* $L(e_\mathcal{A},\sigma,\sigma') = L(\mathcal{A},\sigma,\sigma')$ *for every* $\sigma,\sigma' \in \mathcal{D}_\perp^k$.

The structure of the proof follows of course the standard NFA-regular expressions equivalence, cf. [Sipser, 1997], with all the necessary adjustments to handle transitions induced by $a[c]{\downarrow}I$.

Proof. We prove the first item by induction on the structure of e. In what follows we will identify the vector $\overline{x}$ of variables with the set of registers (i.e. positions) it corresponds to. For example the vector (x_3,x_5) will correspond to the set $I = \{3,5\}$ of registers.

As before, if $(e,w,\sigma) \vdash \sigma'$, we will write $w \in L(e,\sigma,\sigma')$ and similarly if $\mathcal{A}_e = (Q,q_0,F,\delta)$ started with σ accepts w with σ' in the registers, we write $w \in L(\mathcal{A}_e,\sigma,\sigma')$.

- If $e = \emptyset$, then $\mathcal{A}_e = (Q,q_0,F,T)$, where $Q = \{q_0\}$ is the set of states, q_0 is the initial state, $F = \emptyset$ is the set of final states and $T = \emptyset$.

- If $e = \varepsilon$, then $\mathcal{A}_e = (Q,q_0,F,T)$, where $Q = \{q_0\}$ is the set of states, q_0 is the initial state, $F = \{q_0\}$ the set of final states and $T = \emptyset$.

- If $e = a[c]{\downarrow}I$, then $\mathcal{A}_e = (Q,q_0,F,,T)$, where $Q = \{q_0,q_1\}$ is the set of states, q_0 is the initial state, $F = \{q_1\}$ the set of final states and $T = \{(q_0,a,c) \to (I,q_1)\}$.

- If $e = e_1 + e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1,s_1,F_1,T_1)$ and $\mathcal{A}_{e_2} = (Q_2,s_2,F_2,T_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q,q_0,F,T)$, where:

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$, where q_0 is a new state,
- $F = F_1 \cup F_2$,
- To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(q,a,c) \to (I,q') \in T_1 \cup T_2$, where $q = s_1$, or $q = s_2$, we add a transition $(q_0,a,c) \to (I,q')$.

- If $e = e_1 \cdot e_2$ then by the inductive hypothesis we already have automata $\mathcal{A}_{e_1} = (Q_1,s_1,F_1,T_1)$ and $\mathcal{A}_{e_2} = (Q_2,s_2,F_2,T_2)$ with the desired property. The registers of $\mathcal{A}_e$ will be the union of registers of $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$. To obtain the desired automaton $\mathcal{A}_e = (Q,q_0,F,T)$ we distinguish two cases:

  1. If $s_1 \notin F_1$ we set
     - $Q = Q_1 \cup Q_2$,
     - $F = F_2$,
     - $q_0 = s_1$
     - To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(q,a,c) \to (I,q') \in T_1$, where $q' \in F_1$, we add a transition $(q,a,c) \to (I,s_2)$.
  2. If $s_1 \in F_1$ we set
     - $Q = Q_1 \cup Q_2$,
     - $F = \begin{cases} F_2 & \text{if } s_2 \notin F_2 \\ F_1 \cup F_2 & \text{if } s_2 \in F_2 \end{cases}$,
     - $q_0 = s_1$
     - To T we add all transitions from $\mathcal{A}_{e_1}$ and $\mathcal{A}_{e_2}$ and in addition, for every transition $(s_2,a,c) \to (I,q') \in T_2$, we add a transition $(q,a,c) \to (I,q')$, for each $q \in F_1$.

- If $e = e_1^*$ then by the inductive hypothesis we already have the automaton $\mathcal{A}_{e_1} = (Q_1,s_1,F_1,T_1)$ with the desired property. The registers of $\mathcal{A}_e$ will be equal to the registers of $\mathcal{A}_{e_1}$. To obtain the desired automaton we set $\mathcal{A}_e = (Q,q_0,F,T)$, where:
  - $Q = Q_1 \cup \{q_0\}$, where q_0 is a new state,
  - $F = F_1 \cup \{q_0\}$,
  - To T we add all transitions from $\mathcal{A}_{e_1}$ and in addition, for every transition $(s_1,a,c) \to (I,q') \in T_1$, we add a transition $(q_0,a,c) \to (I,q')$ to T. Now for every transition $(q,a,c) \to (I,q') \in T$ (note that we now have transitions from q_0 as well), where $q' \in F_1$, we add $(q,a,c) \to (I,q_0)$ to T

  In all cases it is straightforward to check that the constructed automaton has the desired property. The DLOGSPACE bound follows immediately from the construction.

Next we move onto the second claim of the theorem.

To prove this we will have to introduce generalized register automata (GRA for short) over data words. The difference from usual register automata will be that we allow arrows to be labelled by arbitrary regular expressions over data words. I.e. our arrows are now not labelled only by $a[c]\downarrow I$, but by any regular expression over data words. The transition relation is again called δ and is defined as $\delta \subseteq Q \times \Sigma[x_1, \ldots, x_k] \times Q$. In addition to that we also specify that we have a single initial state with no incoming arrows and a single final state with no outgoing arrows. Note that we also allow ε-transitions.

The only difference is how we define acceptance.

A GRA $\mathcal{A}$ accepts data word w if $w = w_1 \cdot w_2 \cdot \ldots \cdot w_k$(where each w_i is a data word) and there exists a sequence $c_0 = (q_0, 1, \tau_0), \ldots, c_k = (q_k, k+1, \tau_k)$ of configurations of $\mathcal{A}$ on w such that:

1. c_0 is initial,
2. c_k is final,
3. for each i we have $(e_i, w_i, \tau_i) \vdash \tau_{i+1}$(i.e. $w_i \in L(e_i, \tau_i, \tau_{i+1})$), for some e_i such that (q_i, e_i, q_{i+1}) is in the transition relation for $\mathcal{A}$.

We can now prove the equivalence of register automata and regular expressions over data words by mimicking the construction used to prove equivalence between ordinary finite state automata and regular expressions (over strings). Since we use the same construction we will get an exponential blow-up, just like for finite state automata.

Just as in the finite state case we first convert $\mathcal{A}$ into a GRA by adding a new initial state (connected to the old initial state by an ε-arrow) and a new final state (connected to the old end states by incoming ε-arrows). We also assume that this automaton has only a single arrow between every two states (we achieve this by replacing multiple arrows by union of expressions). It is clear that this GRA recognizes the same language of data words as $\mathcal{A}$.

Next we show how to convert this automaton into an equivalent expression. We will use the following recursive procedure which rips out one state at a time from the automaton and stops when we end with only two states (note that this procedure is taken from [Sipser, 1997]).

CONVERT(G)

1. Let n be the number of states of G.

2. If $n = 2$ then G contains only a start state and an end state with a single arrow connecting them. This arrow has an expression R written on it. Return R.

3. If $n > 2$ select any state q_{rip}, different from q_{start} and q_{end} and modify G in the following manner to obtain G' with one less state. The new set of states is $Q' = Q - \{q_{rip}\}$ and for any $q_i \in Q' - \{q_{accept}\}$ and any $q_j \in Q' - \{q_{start}\}$ we define $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) + R_4$, where $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j)$ and $R_4 = \delta(q_i, q_j)$. The initial and final state remain the same.

4. Return CONVERT(G').

We now prove that CONVERT(G) and G recognize the same language of data words. We do so by induction on the number n of states of our GRA G. If $n = 2$ then G has only a single arrow from initial to final state and by definition of acceptance for GRA the expression on this arrow recognizes the same language as G.

Assume now that the claim is true for all automatons with $n-1$ states. Let G be an automaton with n states. We prove that G is equivalent to automaton G' obtained in the step 3 of our CONVERT algorithm. Note that this completes the induction.

To see this assume first that $w \in L(G, \sigma, \sigma')$, i.e. G with initial assignment σ has an accepting run on w ending with σ' in the registers. This means that there exists a sequence of configurations $c_0 = (q_0, 1, \tau_0), \ldots, c_k = (q_k, k, \tau_k)$ such that $w = w_1 w_2 \ldots w_k$, where each w_i is a data word (with possibly more than one symbol), $\tau_0 = \sigma, \tau_k = \sigma'$ and $(\delta(q_{i-1}, q_i), w_i, \tau_{i-1}) \vdash \tau_i$, for $i = 1, \ldots, k$. (Here we used the assumption that we only have a single arrow between any two states).

If none of the states in this run are q_{rip}, then it's also an accepting run in G', so $w \in L(G, \sigma, \sigma')$, since all the arrows present here are also in G'.

If q_{rip} does appear we have the following in our run

$$c_i = (q_i, i, \tau_i), c_{rip} = (q_{rip}, i+1, \tau_{i+1}), \ldots, c_{rip} = (q_{rip}, j-1, \tau_{j-1}), c_j = (q_j, j, \tau_j).$$

If we show how to unfold this to a run in G' we are done (if this appears more than once we apply the same procedure).

Since this is the case we know (by the definition of accepting run) that $(R_1, w_{i+1}, \tau_i) \vdash \tau_{i+1}, (R_2, w_{i+2}, \tau_{i+1}) \vdash \tau_{i+2}, (R_2, w_{i+3}, \tau_{i+2}) \vdash \tau_{i+3}, \ldots, (R_2, w_{j-1}, \tau_{j-2}) \vdash \tau_{j-1}$ and $(R_3, w_j, \tau_{j-1}) \vdash \tau_j$, where $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j)$. Note that this simply means that $((R_1)(R_2)^*(R_3), w_i w_{i+1} \ldots w_j, \sigma) \vdash \sigma'$, so G' can jump from c_i to c_j using only one transition.

Conversely, suppose that $w \in L(G', \sigma, \sigma')$. This means that there is a computation of G' starting with σ and ending with σ' as register assignments. We know that each arrow in G' from q_i to q_j goes either directly (in which case it is already in G) or through q_{rip} (in which

case we use the definition of acceptance by regular expressions to unravel this word into part recognized by G). In either case we get an accepting run of G on w.

To see that this gives the desired result observe that we can always convert register automaton into an equivalent GRA and use CONVERT to obtain a regular expression with memory recognizing the same language. □

Since $L(e) = \bigcup_\sigma L(e, \perp^k, \sigma)$ and $L(\mathcal{A}) = \bigcup_\sigma L(\mathcal{A}, \perp^k, \sigma)$, we obtain:

Corollary 6.2.4. *The classes of languages of data words definable by k-register data word automata, and by regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$ are the same.*

Properties of regular expressions with memory

Closure properties Since Corollary 6.2.4 states that regular expressions with memory and register automata are equivalent, using Fact 6.1.9 we immediately obtain that languages defined by regular expressions with memory are closed under union, intersection, concatenation and Kleene star, but are *not* closed under complement.

Decision problems We start with the nonemptiness problem, i.e., checking whether $L(e) \neq \emptyset$. Since going from expressions to automata is polynomial, we get a PSPACE upper bound (see Fact 6.1.5). Here we also show a matching lower bound.

Proposition 6.2.5. *The nonemptiness problem for regular expressions with memory is* PSPACE*-complete.*

Proof. We prove PSPACE-hardness by doing a reduction from regular automata nonuniversality. This problem requires us to determine, given a finite state automaton $\mathcal{A}$, whether $L(\mathcal{A}) \neq \Sigma^*$.

Assume we are given a regular automaton $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$, where $Q = \{q_1, \ldots, q_n\}$ and $F = \{q_{i_1}, \ldots, q_{i_k}\}$.

Since we are trying to demonstrate nonuniversality of the automaton $\mathcal{A}$ we simulate reachability checking in the powerset automaton for $\overline{\mathcal{A}}$. To do so we designate two distinct data values, t and f, and code each state of the powerset automaton as an n-bit sequence of t/f values, where the ith bit of the sequence is set to t if the state q_i is included in our state of $\overline{\mathcal{A}}$. Since we are checking reachability we will need only to remember the current and the next state of $\overline{\mathcal{A}}$. In what follows we will code those two states using variables $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ and refer to them as the current state tape and the next state tape. Our expression e will code data words that describe successful runs of $\overline{\mathcal{A}}$ by demonstrating how one can move from one state of this automaton to another (as witnessed by their codes in current state tape and next state tape), starting with the initial and ending in a final state.

We will define several expressions and explain their role. We will use two sets of variables, s_1 through s_n and $t_1, \ldots, t_n$ to denote the current state tape and the next state tape. All of these variables will only contain two values, t and f, which are bound in the beginning.

The first expression we need is:

$$\texttt{init} := (a{\downarrow}t) \cdot (a[t^{\neq}]{\downarrow}f) \cdot (a[t^{=}]{\downarrow}s_1) \cdot (a[f^{=}]{\downarrow}s_2) \ldots (a[f^{=}]{\downarrow}s_n).$$

This expression codes two different values as t and f and initializes current state tape to contain encoding of initial state (the one where only the initial state from $\mathcal{A}$ can be reached). That is, a data word is in the language of this expression if and only if it starts with two different data values and continues with n data values that form a sequence in 10^*, where 1 represents the value assigned to t and 0 the one assigned to f.

$$\texttt{end} := a[f^{=} \wedge s_{i_1}^{=}] \cdot a[f^{=} \wedge s_{i_2}^{=}] \cdots a[f^{=} \wedge s_{i_k}^{=}], \text{ where } F = \{q_{i_1}, \ldots, q_{i_k}\}.$$

This expression is used to check that we have reached a state not containing any final state from the original automaton. That is, a data word is in $L(\texttt{end})$ if and only if it consists of k data values, all equal to f and where value stored in s_{i_j} also equals f, for $j = 1 \ldots k$.

Next we define expressions that will reflect updating of the next state tape according to the transition function of $\mathcal{A}$. Assume that $\delta(q_i, b) = \{q_{j_1}, \ldots, q_{j_l}\}$. We define

$$u_{\delta(q_i,b)} := \big((a[t^{=} \wedge s_i^{=}]) \cdot (a[t^{=}]{\downarrow}t_{j_1}) \ldots (a[t^{=}]{\downarrow}t_{j_l})\big) + a[f^{=} \wedge s_i^{=}].$$

Also, if $\delta(q_i, b) = \emptyset$ we simply put $u_{\delta(q_i,b)} := \varepsilon$.

This expression will be used to update the next state tape by writing true to corresponding variables if the state q_i is tagged with t on the current state tape (and thus contained in the current state of $\overline{\mathcal{A}}$). If it is false we skip the update.

Since we have to define update according to all transitions from all the states corresponding to chosen letter we get:

$$\texttt{update} := \bigvee_{b \in \Sigma} \bigwedge_{q_i \in Q} u_{\delta(q_i,b)}.$$

This simply states that we non deterministically pick the next symbol of the word we are guessing and move to the next state accordingly.

We still have to ensure that the tapes are copied at the beginning and end of each step, so we define:

$$\texttt{step} := \big((a[f^{=}]{\downarrow}t_1) \ldots (a[f^{=}]{\downarrow}t_n)\big) \cdot \texttt{update} \cdot \big((a[t_1^{=}]{\downarrow}s_1) \ldots (a[t_n^{=}]{\downarrow}s_n)\big).$$

This simply initializes the next state tape at the beginning of each step, proceeds with the update and copies the next state tape to the current state tape.

Finally we have

$$e := \texttt{init} \cdot (\texttt{step})^* \cdot \texttt{end}.$$

We claim that for $L(e) \neq \emptyset$ if and only if $L(\mathcal{A}) \neq \Sigma^*$.

Assume first that $L(\mathcal{A}) \neq \Sigma^*$. This means that there is a path from the initial to the final state in the powerset automaton for $\overline{\mathcal{A}}$. That is, there is a word w from Σ^* not in the language of $\mathcal{A}$. This path can in turn be described by pairs of assignment of values t/f to the current state tape and the next state tape, where each transition is witnessed by the corresponding letter of the alphabet. But then the word that belongs to $L(e)$ is the one that first initializes the stable tape (i.e. the variables $s_1, \ldots, s_n$) to initial state of the powerset automaton, then runs the updates of the tape according to w and finally ends in a state where all variable corresponding to end states of $\mathcal{A}$ are tagged f.

Conversely, each word from s to t in $L(e)$ corresponds to a run of the powerset automaton for $\overline{\mathcal{A}}$. That is, the part of word corresponding to `init` sets the initial state. Then the part of this word that corresponds to $\texttt{step}^*$ corresponds to updating our tapes in a way that properly codes one step of powerset automaton. Finally, `end` denotes that we have reached a state where all end states of $\mathcal{A}$ have been tagged by f, thus, an accepting state for $\overline{\mathcal{A}}$. □

Next we move to the membership problem, i.e., checking whether $w \in L(e)$. Again, since e can be translated efficiently into an equivalent automaton $\mathcal{A}_e$, Fact 6.1.6 gives an NP upper bound. We can prove a matching lower bound as well:

Proposition 6.2.6. *The membership problem for regular expressions with memory is* NP*-complete.*

Proof. For the lower bound we do a reduction from 3-SAT.

Let $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \ldots \wedge (a_k \vee b_k \vee c_k)$, be an arbitrary 3-CNF formula. We will construct a data word w and a regular expression with memory e, both of length linear in the length of φ, such that φ is satisfiable if and only if $w \in L(e)$.

Let $x_1, x_2, \ldots, x_n$ be all the variables occurring in φ. We define w as the following data word:

$$w = \left(\binom{a}{0}\binom{b}{1}\right)^n \left(\binom{a_1}{d_{a_1}}\binom{b_1}{d_{b_1}}\binom{c_1}{d_{c_1}}\right) \cdots \left(\binom{a_k}{d_{a_k}}\binom{b_k}{d_{b_k}}\binom{c_k}{d_{c_k}}\right),$$

where $d_{a_i} = 1$, if $a_i = x_j$, for some $j \in \{1, \ldots n\}$ and 0, if $a_i = \overline{x_j}$ and similarly for d_{b_i}, d_{c_i} (note that every a_i, b_i, c_i is of the form x_j, or $\overline{x_j}$, so this is well defined).

Also note that we are using a_i, b_i, c_i both for literals in φ and for letters of our finite alphabet, but this should not arise any confusion. The idea behind this data word is that with the first part that corresponds to the variables, i.e. with $(\binom{a}{0}\binom{b}{1})^n$, we guess a satisfying assignment and the next part corresponds to each conjunct in φ and its data value is set such that if we stop at any point for comparison we get a true literal in this conjunct.

We now define e as the following regular expression with memory:

$$e = (a{\downarrow}x_1 + ab{\downarrow}x_1) \cdot b^* \cdot (a{\downarrow}x_2 + ab{\downarrow}x_2) \cdot b^* \cdot (a{\downarrow}x_3 + ab{\downarrow}x_3) \cdots \\ b^* \cdot (a{\downarrow}x_n + ab{\downarrow}x_n) \cdot b^* \cdot \text{clause}_1 \cdot \text{clause}_2 \ldots \text{clause}_k,$$

where each clause_i corresponds to the i-th conjunct of φ in the following manner.

If ith conjunct uses variables $x_{j_1}, x_{j_2}, x_{j_3}$(possibly with repetitions), then

$$\text{clause}_i = a_i[x_{j_1}^{=}] \cdot b_i \cdot c_i + a_i \cdot b_i[x_{j_2}^{=}] \cdot c_i + a_i \cdot b_i \cdot c_i[x_{j_3}^{=}].$$

We now prove that φ is satisfiable if and only if $w \in L(e)$.

Assume first that φ is satisfiable. Then there's a way to assign a value to each x_i such that for every conjunct in φ at least one literal is true. This means that we can traverse the first part of w to chose the corresponding values for variables bounded in e. Now with this choice we can make one of the literals in each conjunct true, so we can traverse every clause_i using one of the tree possibilities.

Assume now that $w \in L(e)$. This means that after choosing the data values for variables (and thus a valuation for φ, since all data values are either 0 or 1), we are able to traverse the second part of w using these values. This means that for every clause_i there is a letter after which the data value is the same as the one bounded to the corresponding variable. Since data values in the second part of w correspond to literal in the corresponding conjunct of φ to evaluate to 1, we know that this valuation satisfies our formula φ. □

Finally, using Theorem 6.2.3 and Fact 6.1.8 we also get the following result about universality and containment.

Corollary 6.2.7. *Universality and containment problems are undecidable for regular expressions with memory.*

6.3 Regular expressions with binding

Here we redefine regular expressions with binding to work over data words instead of data paths. As already mentioned in Section 4.3, expressions with binding were originally developed as a graph querying formalism that restricts the use of variables in regular expressions with memory by imposing proper scoping rules. The idea here is to use variables to store data values and then compare them using conditions. The storing of a value, however, will bind it only to the scope of the variable used, unlike in regular expressions with memory.

Conditions are defined in the same manner as in Section 6.2. Next we define regular expressions with binding.

Definition 6.3.1. *Let Σ be a finite alphabet and $\{x_1,\dots,x_k\}$ a finite set of variables.* Regular expressions with binding *(REWB) over $\Sigma[x_1,\dots,x_k]$ are defined inductively as follows:*

$$r \;:=\; \varepsilon \;\mid\; a \;\mid\; a[c] \;\mid\; r+r \;\mid\; r\cdot r \;\mid\; r^* \;\mid\; a\downarrow_{x_i}.\{r\} \tag{6.1}$$

where $a\in\Sigma$ and c is a condition in $\mathcal{C}_k$.

A variable x_i is bound if it occurs in the scope of some $\downarrow_{x_i}$ operator and free otherwise. More precisely, free variables of an expression are defined inductively: ε and a have no free variables, in $a[c]$ all variables occurring in c are free, in r_1+r_2 and $r_1\cdot r_2$ the free variables are those of r_1 and r_2, the free variables of r^* are those of r, and the free variables of $a\downarrow_{x_i}.\{r\}$ are those of r except x_i. We will write $r(x_1,\dots,x_l)$ if $x_1,\dots,x_l$ are the free variables in r.

A valuation on the variables $x_1,\dots,x_k$ is a partial function $\nu:\{x_1,\dots,x_k\}\mapsto\mathcal{D}$. We denote by $\mathcal{F}(x_1,\dots,x_k)$ the set of all valuations on $x_1,\dots,x_k$. For a valuation ν, we write $\nu[x_i\leftarrow d]$ to denote the valuation ν' obtained by fixing $\nu'(x_i)=d$ and $\nu'(x)=\nu(x)$ for all other $x\neq x_i$. Likewise, we write $\nu[\overline{x}\leftarrow\overline{d}]$ for a simultaneous substitution of values from $\overline{d}=(d_1,\dots,d_l)$ for variables $\overline{x}=(x_1,\dots,x_l)$. Also notation $\nu(\overline{x})=\overline{d}$ means that $\nu(x_i)=d_i$ for all $i\leq l$.

Semantics Let $r(\overline{x})$ be an REWB over $\Sigma[x_1,\dots,x_k]$. A valuation $\nu\in\mathcal{F}(x_1,\dots,x_k)$ is compatible with r, if $\nu(\overline{x})$ is defined.

A regular expression $r(\overline{x})$ over $\Sigma[x_1,\dots,x_k]$ and a valuation $\nu\in\mathcal{F}(x_1,\dots,x_k)$ compatible with r define a language $L(r,\nu)$ of data words as follows.

- If $r=a$ and $a\in\Sigma$, then $L(r,\nu)=\{\binom{a}{d}\mid d\in\mathbb{N}\}$.
- If $r=a[c]$, then $L(r,\nu)=\{\binom{a}{d}\mid d,\nu\models c\}$.
- If $r=r_1+r_2$, then $L(r,\nu)=L(r_1,\nu)\cup L(r_2,\nu)$.
- If $r=r_1\cdot r_2$, then $L(r,\nu)=L(r_1,\nu)\cdot L(r_2,\nu)$.
- If $r=r_1^*$, then $L(r,\nu)=L(r_1,\nu)^*$.
- If $r=a\downarrow_{x_i}.\{r_1\}$, then $L(r,\nu)=\bigcup_{d\in\mathcal{D}}\left\{\binom{a}{d}\right\}\cdot L(r_1,\nu[x_i\leftarrow d])$.

A REWB r defines a language of data words as follows.

$$L(r) \;=\; \bigcup_{\nu\text{ compatible with }r} L(r,\nu).$$

In particular, if r is without free variables, then $L(r)=L(r,\emptyset)$. We will call such REWBs *closed*.

Example 6.3.2. *We list several examples of languages expressible with our expressions. In all cases below we have a singleton alphabet $\Sigma=\{a\}$.*

- *The language that consists of data words where the data value in the first position is different from the others is given by:* $a\downarrow_x.\{(a[x^{\neq}])^*\}$.
- *The language that consists of data words where the data values in the first and the last position are the same is given by:* $a\downarrow_x.\{a^*\cdot a[x^{=}]\}$.
- *The language that consists of data words where there are two positions with the same data value:* $a^*\cdot a\downarrow_x.\{a^*\cdot a[x^{=}]\}\cdot a^*$.

Note that in REWBs in the above example the conditions are very simple: they are either $x^{=}$ or $x^{\neq}$. We will call such expressions *simple* REWBs.

We shall also consider *positive* REWBs where negation and inequality are disallowed in conditions. That is, all the conditions c are constructed using the following syntax: $c := \mathtt{tt} \mid x_i^{=} \mid c\wedge c \mid c\vee c$,, where $1\leq i\leq k$.

Closure properties and connection with register automata

As mentioned before, regular expressions with memory have a similar syntax but rather different semantics than REWBs. They are built using $a\downarrow_x$, concatenation, union and Kleene star. That is, no binding is introduced with $a\downarrow_x$; rather it directly matches the operation of putting a value in a register. In contrast, REWBs use proper bindings of variables; expression $a\downarrow_x$ appears only in the context $a\downarrow_x.\{r\}$ where it binds x inside the expression r only. Theorem 6.2.3 states that expressions with memory and register automata are one and the same in terms of expressive power. Here we show that REWBs, on the other hand, are strictly weaker. Therefore, proper binding of variables comes with a cost – albeit small – in terms of expressiveness.

Theorem 6.3.3. *The class of languages defined by REWBs is strictly contained in the class of languages accepted by register automata.*

That the class of languages defined by REWBs is contained in the class of languages defined by register automata can be proved by using a similar inductive construction as in Theorem 6.2.3.

To show that the containment is strict we need to examine closure properties of REWB languages.

Closure properties It follows from the definition that regular expressions with binding are closed under union, concatenation and Kleene star. Next we show they are not closed under complement.

Proposition 6.3.4. *The class of languages definable by regular expressions with binding is not closed under complement.*

Proof. To see that they are not closed under complement, recall from Example 6.3.2 that the expression $a^* \cdot a \downarrow_x .\{a^* \cdot a[x^=]\} \cdot a^*$ defines the set of all data words with two positions with the same data value. The complement of this language, where all data values are different is well known not to be definable by register automata [Kaminski and Francez, 1994]. □

We also show that REWB languages are not closed under intersection. The proof of this fact will also imply Theorem 6.3.3.

Theorem 6.3.5. *REWB languages are not closed under intersection.*

To prove this we define two languages, L_1 and L_2, both easily definable by a regular expression with binding, but such that their intersection is not REWB definable.

Let L_1 be the language consists of data words of the form:

$$\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3}\binom{a}{d_4}\binom{a}{d_5}\binom{a}{d_6}\binom{a}{d_7}\binom{a}{d_8}\cdots\cdots\binom{a}{d_{4n}}$$

where $d_2 = d_5, d_6 = d_9, \ldots, d_{4n-6} = d_{4n-3}$.

Let L_2 be the language as above, but $d_4 = d_7, d_8 = d_{11}, \ldots, d_{4n-4} = d_{4n-1}$.

In particular, $L_1 \cap L_2$ is the language consisting of data words of the form:

$$\tbinom{a}{d_1}\tbinom{a}{d_2}\tbinom{a}{e_1}\tbinom{a}{e_2}\tbinom{a}{d_2}\tbinom{a}{d_3}\tbinom{a}{e_2}\tbinom{a}{e_3}\cdots\cdots\tbinom{a}{d_{m-2}}\tbinom{a}{d_{m-1}}\tbinom{a}{e_{m-2}}\tbinom{a}{e_{m-1}}\tbinom{a}{d_{m-1}}\tbinom{a}{d_m}\tbinom{a}{e_{m-1}}\tbinom{a}{e_m}$$

Both L_1 and L_2 are REWB languages. We are now going to show the following.

Lemma 6.3.6. *$L_1 \cap L_2$ is not a REWB language.*

Note that for simplicity we prove the theorem for the case of simple REWBs. It is straightforward to see that the same proof works in the case of REWBs that use multiple comparisons in one condition.

The proof is rather technical and will require a few auxiliary notions. Let r be an REWB over $\Sigma[x_1, \ldots, x_k]$. A *derivation tree t* with respect to r is a tree whose internal nodes are labeled with (r', ν) where r' is an subexpression of r and $\nu \in \mathcal{F}(x_1, \ldots, x_k)$ constructed as follows. The root node is labeled with $(e, \emptyset)$. The other nodes are labeled as follows. For a node u labeled with (r', ν), its children are labeled as follows.

- If $r' = a$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ for some $d \in \mathcal{D}$.
- If $r' = a[\varphi]$, then u has only one child: a leaf node labeled with $\binom{a}{d}$ such that $d, \nu \models \varphi$.
- If $r' = r_1 + r_2$, then u has only one child: a leaf node labeled with either (r_1, ν) or (r_2, ν).
- If $r' = r_1 \cdot r_2$, then u has only two children: the left child is labeled with (r_1, ν) and the right child is labeled with (r_2, ν).

- If $r' = r_1^*$, then u has either only one child: a leaf node labeled with ε; or at least one child labeled with (r_1, ν).
- If $r' = a \downarrow_x .\{r_1\}$, then u has only two children: the left child is labeled with $\binom{a}{d}$ and the right child is labeled with $(r_1, \nu[x \leftarrow d])$, for some data value $d \in \mathcal{D}$.

A derivation tree t defines a data word $w(t)$ as the word read on the leaf nodes of t from left to right.

Proposition 6.3.7. *For every REWB r, the following holds. A data word $w \in L(r, \emptyset)$ if and only if there exists a derivation tree t w.r.t. r such that $w = w(t)$.*

Proof. We start with the "only if" direction. Suppose that $w \in L(r, \emptyset)$. By induction on the length of e, we can construct the derivation tree t such that $w = w(t)$. It is a rather straightforward induction, where the induction step is based on the recursive definition of REWB, where r is either a, $a[x^{=}]$, $a[x^{\neq}]$, $r_1 + r_2$, $r_1 \cdot r_2$, r_1^* or $a \downarrow_x .\{r_1\}$.

Now we prove the "if" direction.

For a node u in a derivation tree t, the word induced by the node u is the subword made up of the leaf nodes in the subtree rooted at u. We denote such subword by $w_u(t)$.

We are going to show that for every node u in t, if u is labeled with (r', ν), then $w_u(t) \in L(r', \nu)$. This can be proved by induction on the *height* of the node u, which is defined as follows.

- The height of a leaf node is 0.
- The height of a node u is the maximum between the heights of its children nodes plus one.

It is a rather straightforward induction, where the base case is the nodes with zero height and the induction step is carried on nodes of height h with the induction hypothesis assumed to hold on nodes of height $< h$. □

Suppose $w(t) = w_1 w_u(t) w_2$, the *index pair* of the node u is the pair of integers (i, j) such that $i = \text{length}(w_1) + 1$ and $j = \text{length}(w_1 w_u(t))$.

A derivation tree t induces a binary relation R_t as follows.

$$R_t = \{(i, j) \mid (i, j) \text{ is the index pair of a node } u \text{ in } t \text{ labeled with } a \downarrow_{x_l} .\{r'\} \quad \}.$$

Note that R_t is a partial function from the set $\{1, \ldots, \text{length}(w(t))\}$ to itself, where if $R_t(i)$ is defined, then $i < R_t(i)$.

For a pair $(i, j) \in R_t$, we say that the variable x is associated with (i, j), if (i, j) is the index pair of a node u in t labeled with a label of the form $a \downarrow_x .\{r'\}$. Two binary tuples (i, j) and (i', j'), where $i < j$ and $i' < j'$, *cross each other* if either $i < i' < j < j'$ or $i' < i < j' < j$.

Proposition 6.3.8. *For any derivation tree t, the binary relation R_t induced by it does not contain any two pairs (i,j) and (i',j') that cross each other.*

Proof. Suppose $(i,j),(i',j') \in R_t$. Then let u and u' be the nodes whose index pairs are (i,j) and (i',j'), respectively. There are two cases.

- The nodes u and u' are descendants of each other.
  Suppose u is a descendant of u'. Then, we have $i' < i < j < j'$.
- The nodes u and u' are not descendants of each other.
  Suppose the node u' is on the right side of u, that is, $w_{u'}(t)$ is on the right side of $w_u(t)$ in w. Then we have $i < j < i' < j'$.

In either case (i,j) and (i',j') do not cross each other. This completes the proof of our claim. □

Now we are ready to show that $L_1 \cap L_2$ is not defined by any REWB. Suppose to the contrary that there is an REWB r over $\Sigma[x_1,\ldots,x_k]$ such that $L(r) = L_1 \cap L_2$, where $\Sigma = \{a\}$. Consider the following word w, where $m = k+2$:

$$w := \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{e_1}\binom{a}{e_2}\cdots\cdots\cdots \binom{a}{d_{m-2}}\binom{a}{d_{m-1}}\binom{a}{e_{m-2}}\binom{a}{e_{m-1}}\binom{a}{d_{m-1}}\binom{a}{d_m}\binom{a}{e_{m-1}}\binom{a}{e_m}$$

where $d_0, d_1, \ldots, d_m, e_0, e_1, \ldots, e_m$ are pairwise different.

Let t be the derivation tree of w. Consider the binary relation R_t and the following sets A and B.

$$\begin{aligned} A &= \{2, 6, 10, \ldots, 4m-6\} \\ B &= \{4, 8, 12, \ldots, 4m-4\} \end{aligned}$$

That is, the set A contains the first positions of the data values $d_1,\ldots,d_{m-1}$s, and the set B the first positions of the data values $e_1,\ldots,e_{m-1}$s.

Claim 6.3.9. *The relation R_t is a function on $A \cup B$. That is, for every $h \in A \cup B$, there is h' such that $(h,h') \in R_t$.*

Proof. Suppose there exists $h \in A \cup B$ such that $R_t(h)$ is not defined. Assume that $h \in A$ and l be such that $h = 4l-2$. If $R_t(h)$ is not defined, then for any valuation ν found in the nodes in t, $d_l \notin \text{Image}(\nu)$. So, the word

$$w'' = \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{d_{l-1}}\binom{a}{f}\binom{a}{e_{l-1}}\binom{a}{e_l}\binom{a}{d_l}\binom{a}{d_{l+1}}\cdots\cdots$$

is also in $L(r)$, where f is a new data value. That is, the word w'' is obtained by replacing the first appearance of d_l with f. Now $w'' \notin L_1 \cap L_2$, hence, contradicts the fact that $L(r) = L_1 \cap L_2$. The same reasoning goes for the case if $h \in B$. This completes the proof of our claim. □

Remark 6. *Without loss of generality, we can assume that each variable in the REWB r is introduced only once. Otherwise, we can rename the variable.*

Claim 6.3.10. *There exist $(h_1,h_2),(h_1',h_2') \in R_t$ such that $h_1 < h_2 < h_1' < h_2'$ and $h_1,h_1' \in A$ and both $(h_1,h_2),(h_1',h_2')$ have the same associated variable.*

Proof. The cardinality $|A| = k+1$. So there exists a variable $x \in \{x_1,\ldots,x_k\}$ and $(h_1,h_2),(h_1',h_2') \in R_t$ such that $(h_1,h_2),(h_1',h_2')$ are associated with the variable x. By Remark 6, no variable is written twice in e, so the nodes u,u' associated with $(h_1,h_2),(h_1',h_2')$ are not descendants of each other, so we have $h_1 < h_2 < h_1' < h_2'$, or $h_1' < h_2' < h_1 < h_2$. This completes the proof of our claim. □

Claim 6.3.11 below immediately implies that Lemma 6.3.6.

Claim 6.3.11. *There exists a word $w'' \notin L_1 \cap L_2$, but $w'' \in L(r)$.*

Proof. The word w'' is constructed from the word w. By Claim 6.3.10, there exist $(h_1,h_2),(h_1',h_2') \in R_t$ such that $h_1 < h_2 < h_1' < h_2'$ and $h_1,h_1' \in A$ and both h_1,h_1' have the same associated variable.

By definition of the language $L_1 \cap L_2$, between h_1 and h_1', there exists an index $l \in B$ such that $h_1 < l < h_1'$. (Recall that the set A contains the first positions of the data values $d_1,\ldots,d_{m-1}$s, and the set B the first positions of the data values $e_1,\ldots,e_{m-1}$s.)

Let h be the maximum of such indices. The index h is not the index of the last e, hence $R_t(h)$ exists and $R_t(h) < h_2$, by Proposition 6.3.8. Now the data value in $R_t(h)$ is different from the data value in position h. To get w'', we change the data value in the position h with a new data value f, and it will not change the acceptance of the word w'' by the REWB r.

However, the word w''

$$w'' \quad = \quad \binom{a}{d_0}\binom{a}{d_1}\binom{a}{e_0}\binom{a}{e_1}\cdots\cdots\binom{a}{e_{l-1}}\binom{a}{f}\cdots\binom{a}{e_l}\binom{a}{e_{l+1}}\cdots\cdots$$

is not in $L_1 \cap L_2$, by definition. Thus, this completes the proof of our claim. □

This completes our proof of Lemma 6.3.6.

Since both L_1 and L_2 are easily definable by a REWB using only one variable, this completes the proof of Theorem 6.3.5.

As a corollary of this we also get the proof of Theorem 6.3.3. We note that the separating example is rather intricate, and certainly not a natural language one would think of. In fact, all natural languages definable with register automata that we used here as examples – and many more, especially those suitable for graph querying – are definable by REWBs.

Decision problems

Nonemptiness and membership Recall that for register automata, the nonemptiness problem is PSPACE-complete (and the same bound applied to regular expressions with memory). By introducing proper binding we lose some expressiveness and yet can lower the complexity of the problem to NP.

Note that standard nonemptiness checks if the language of a *closed* REWB is empty. More generally, one can ask if $L(r,\nu) \neq \emptyset$ for a REWB r and a compatible valuation ν.

Theorem 6.3.12. *The nonemptiness problem for REWBs is NP-complete.*

Proof. In order to prove the NP-upper bound from the theorem we will first show that if there is a word accepted by a REWB, then there is also a word accepted that is no longer than the REWB itself.

Proposition 6.3.13. *For every REWB r over $\Sigma[x_1,\ldots,x_k]$ and every valuation ν compatible with r, if $L(r,\nu) \neq \emptyset$, then there exists a data word $w \in L(r,\nu)$ of length $O(|r|)$.*

Proof. The proof is by induction on the length of r. The basis is when the length of r is 1. There are two cases: $a[c]$ and a; and it is trivial that our proposition holds.

Let r be an REWB and ν a valuation compatible with r. For the induction hypothesis, we assume that our proposition holds for all REWBs of shorter length than r. For the induction step, we prove our proposition for r. There are four cases.

- Case 1: $r = r_1 + r_2$.
  If $L(r,\nu) \neq \emptyset$, then by the induction hypothesis, either $L(r_1,\nu)$ or $L(r_2,\nu)$ are not empty. So, either
  - there exists $w_1 \in L(r_1,\nu)$ such that $|w_1| = O(|r_1|)$; or
  - there exists $w_2 \in L(r_2,\nu)$ such that $|w_2| = O(|r_2|)$.

  Thus, by definition, there exists $w \in L(r,\nu)$ such that $|w| = O(|r|)$.
- Case 2: $r = r_1 \cdot r_2$.
  If $L(r,\nu) \neq \emptyset$, then by the definition, $L(r_1,\nu)$ and $L(r_2,\nu)$ are not empty. So by the induction hypothesis
  - there exists $w_1 \in L(r_1,\nu)$ such that $|w_1| = O(|r_1|)$; and
  - there exists $w_2 \in L(r_2,\nu)$ such that $|w_2| = O(|r_2|)$.

  Thus, by definition, $w_1 \cdot w_2 \in L(r,\nu)$ and $|w_1 \cdot w_2| = O(|r|)$.
- Case 3: $r = (r_1)^*$.
  This case is trivial since $\varepsilon \in L(r,\nu)$.

- Case 4: $r = a\downarrow_{x_i}.\{r_1\}$.

  If $L(r,\nu) \neq \emptyset$, then by the definition, $L(r_1, \nu[x_i \leftarrow d])$ is not empty, for some data value d. By the induction hypothesis, there exists $w_1 \in L(r_1, \nu[x_i \leftarrow d])$ such that $|w_1| = O(|r_1|)$. By definition, $\binom{a}{d} w_1 \in L(r,\nu)$.

This completes the proof of Proposition 6.3.13. □

The NP membership now follows from Proposition 6.3.13, where given a REWB r, we simply guess a data word $w \in L(r)$ of length $O(|r|)$. The verification that $w \in L(r)$ can also be done in NP (Proposition 6.3.15).

Note that the data values here can be made small as well. This follows from the fact that in a word accepted by a register automaton one can replace the data values with the ones from the set $1,\ldots k+1$, where k is the number of registers (see Lemma 6.1.3), while retaining the acceptance condition. Thus we can always assume that the values appearing in our word are not bigger than the number of variables in our expression plus one.

We prove NP hardness via a reduction from 3-SAT.

Assume that $\varphi = (\ell_{1,1} \vee \ell_{1,2} \vee \ell_{1,3}) \wedge \cdots \wedge (\ell_{n,1} \vee \ell_{n,2} \vee \ell_{n,3})$ is the given 3-CNF formula, where each $\ell_{i,j}$ is a literal. Let $x_1,\ldots x_k$ denote the variables occurring in φ. We say that the literal $\ell_{i,j}$ is negative, if it is a negation of a variable. Otherwise, we call it a positive literal.

We will define a REWB r over $\Sigma[y_1,z_1,y_2,z_2,\ldots,y_k,z_k]$ of length $O(n)$ such that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Let r be the following REWB.

$$\begin{aligned} r \;:=\; & a\downarrow_{y_1}.\{a\downarrow_{z_1}.\{a\downarrow_{y_2}.\{a\downarrow_{z_2}.\{\cdots\{a\downarrow_{y_k}.\{a\downarrow_{z_k}.\{ \\ & (r_{1,1}+r_{1,2}+r_{1,3})\cdots(r_{n,1}+r_{n,2}+r_{n,3})\}\}\ldots\}, \end{aligned}$$

$$r_{i,j} \;:=\; \begin{cases} b[y_k^{=} \wedge z_k^{=}] & \text{if } \ell_{i,j} = x_k \\ b[y_k^{=} \wedge z_k^{\neq}] + b[z_k^{=} \wedge y_k^{\neq}] & \text{if } \ell_{i,j} = \neg x_k \end{cases}$$

Obviously, $|r| = O(n)$. We are going to prove that φ is satisfiable if and only if $L(r) \neq \emptyset$.

Assume first that φ is satisfiable. Then there is an assignment $f : \{x_1,\ldots,x_k\} \mapsto \{0,1\}$ making φ true. We define the evaluation $\nu : \{y_1,z_1,\ldots y_n,z_n\} \mapsto \{0,1\}$ as follows.

- If $f(x_i) = 1$, then $\nu(y_i) = \nu(z_i) = 1$.
- If $f(x_i) = 0$, then $\nu(y_i) = 0$ and $\nu(z_i) = 1$.

We define the following data word.

$$w \;:=\; \binom{a}{\nu(y_1)}\binom{a}{\nu(z_1)}\cdots\binom{a}{\nu(y_k)}\binom{a}{\nu(z_k)}\underbrace{\binom{b}{1}\cdots\binom{b}{1}}_{n \text{ times}}$$

To see that $w \in L(r)$, we observe that the first $2k$ labels are parsed to bind values $y_1, z_1, \ldots y_k, z_k$ to corresponding values determined by ν. To parse the remaining $\binom{b}{1} \cdots \binom{b}{1}$, we observe that for each $i \in \{1, \ldots, n\}$, $\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$ is true according to the assignment f if and only if $\binom{b}{1} \in L(r_{i,1} + r_{i,2} + r_{i,3}, \nu)$.

Conversely, assume that $L(r) \neq \emptyset$. Let

$$w = \binom{a}{d_{y_1}} \binom{a}{d_{z_1}} \cdots \binom{a}{d_{y_k}} \binom{a}{d_{z_k}} \binom{b}{d_1} \cdots \binom{b}{d_n} \in L(r).$$

We define the following assignment $f : \{x_1, \ldots, x_k\} \mapsto \{0, 1\}$.

$$f(x_i) \quad = \quad \begin{cases} 1 & \text{if } d_{y_i} = d_{z_i} \\ 0 & \text{if } d_{y_i} \neq d_{z_i} \end{cases}$$

We are going to show that f is a satisfying assignment for φ. Now since $w \in L(r)$, we have

$$\binom{b}{d_1} \cdots \binom{b}{d_n} \quad \in \quad L((r_{1,1} + r_{1,2} + r_{1,3}) \cdots (r_{n,1} + r_{n,2} + r_{n,3}), \nu),$$

where $\nu(y_i) = d_{y_i}$ and $\nu(z_i) = d_{z_i}$. In particular, we have for every $j = 1, \ldots, n$,

$$\binom{b}{d_j} \quad \in \quad L(r_{j,1} + r_{j,2} + r_{j,3}, \nu).$$

W.l.o.g, assume that $\binom{b}{d_j} \in L(r_{j,1})$. There are two cases.

- If $r_{j,1} = b[y_i^{=} \wedge z_i^{=}]$, then by definition, $\ell_{j,1} = x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.
- If $r_{j,1} = b[y_i^{=} \wedge z_i^{\neq}] + b[z_i^{=} \wedge y_i^{\neq}]$, then by definition, $\ell_{j,1} = \neg x_i$, hence the clause $\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ is true under the assignment f.

Thus, the assignment f is a satisfying assignment for the formula φ. This completes the proof of Theorem 6.3.12. □

Note that for simple and positive REWBs the problem trivializes.

Proposition 6.3.14.
- *For every simple REWB r over $\Sigma[x_1, \ldots, x_k]$, and for every valuation ν compatible with r, we have $L(r, \nu) \neq \emptyset$.*
- *For every positive REWB r over $\Sigma[x_1, \ldots, x_k]$, there is a valuation ν such that $L(r, \nu) \neq \emptyset$.*

For membership we only have the upper bound.

Proposition 6.3.15. *Membership problem for REWBs is in* NP.

This immediately follows from Theorem 6.3.3 and the bound for register automata.

Containment and universality Next we examine the containment and universality problems for REWBs. It turns out that both are undecidable. In fact, we can show an even stronger statement, that *universality* of simple REWBs that use just a single variable is already undecidable.

Theorem 6.3.16. *Universality for one-variable REWBs is undecidable. In particular general universality and containment are also undecidable.*

Proof. We are first going to prove that given an REWB r over $\Sigma[x_1,\ldots,x_k]$, checking whether $L(e) = (\Sigma \times \mathcal{D})^*$ is undecidable. This immediately implies that given r_1, r_2, checking whether $L(r_1) \subseteq L(r_2)$ is undecidable, hence, the second item of our theorem.

The proof is similar to the proof of the universality of register automata in [Neven et al., 2004]. The reduction is via Post Correspondence Problem (PCP), which is defined as follows. An instance of PCP is a set of pairs of strings

$$I = \{(u_1, v_1), \ldots, (u_n, v_n)\},$$

where $u_i, v_i \in \Sigma^*$. A solution of the instance I is a sequence $l_1, \ldots, l_m$ such that $u_{l_1} \cdots u_{l_m} = v_{l_1} \cdots v_{l_m}$.

Let $\$, \#$ be two special symbols not in Σ. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} w_2$ over $\Sigma \cup \{\$, \#\}$, where

$$\begin{aligned} w_1 &= \tbinom{\$}{e_1}\tbinom{a_1}{d_1}\cdots\tbinom{a_{\ell_1}}{d_{\ell_1}}\tbinom{\$}{e_2}\tbinom{a_{\ell_1+1}}{d_{\ell_1+1}}\cdots\tbinom{a_{\ell_1+\ell_2}}{d_{\ell_1+\ell_2}}\tbinom{\$}{e_3}\cdots\cdots\tbinom{\$}{e_m}\tbinom{a_{\ell_1+\cdots+\ell_{m-1}}}{d_{\ell_1+\cdots+\ell_{m-1}}}\cdots\tbinom{a_\ell}{d_\ell} \\ w_2 &= \tbinom{\$}{g_1}\tbinom{b_1}{f_1}\cdots\tbinom{b_{\ell_1}}{f_{\ell_1}}\tbinom{\$}{g_2}\tbinom{b_{\ell_1+1}}{f_{\ell_1+1}}\cdots\tbinom{b_{\ell_1+\ell_2}}{f_{\ell_1+\ell_2}}\tbinom{\$}{g_3}\cdots\cdots\tbinom{\$}{g_m}\tbinom{b_{\ell_1+\cdots+\ell_{m-1}}}{f_{\ell_1+\cdots+\ell_{m-1}}}\cdots\tbinom{b_\ell}{f_\ell} \end{aligned}$$

where $\ell = \ell_1 + \ell_2 + \cdots + \ell_m$, and

(C1) The symbol # appears only once.

(C2) $\mathsf{Proj}_\Sigma(w_1) \in (\$ \cdot u_1 + \cdots + \$ \cdot u_n)^*$.

(C3) $\mathsf{Proj}_\Sigma(w_2) \in (\$ \cdot v_1 + \cdots + \$ \cdot v_n)^*$.

(C4) The data values e_i's and d_i's are pairwise different.

(C5) The data values g_i's and f_i's are pairwise different.

(C6) $e_1 = g_1$ and $e_m = g_m$.

(C7) $d_1 = f_1$ and $d_{\ell_m} = f_{\ell_m}$.

(C8) For all $i \in \{1,\ldots,m-1\}$, there exists $j \in \{1,\ldots,m-1\}$ such that $e_i = g_j$ and $e_{i+1} = g_{j+1}$.

(C9) For all $i \in \{1,\ldots,\ell_m - 1\}$, there exists $j \in \{1,\ldots,\ell_m - 1\}$ such that $d_i = f_j$ and $d_{i+1} = f_{j+1}$.

(C10) For all $i, j \in \{1,\ldots,\ell_m\}$, if $d_i = f_j$, then $a_i = b_j$.

(C11) For all $i, j \in \{1,\ldots,m\}$, if $e_i = g_j$, then $(a_{\ell_{i-1}+1} \cdots a_{\ell_i}, b_{\ell_{j-1}+1} \cdots b_{\ell_j}) \in I$.

Now it is straightforward to show that there exists a solution to the PCP instance I if and only if there exists a data word over $\Sigma \cup \{\$, \#\}$ that satisfies Conditions (C1)–(C11) above.

We now construct an REWB e over $\Sigma_1[x_1, \ldots, x_k]$ where $\Sigma_1 = \Sigma \cup \{\$, \#\}$ that accepts a data word w that does not satisfies at least one of the Conditions (C1) to (C11) above. Such REWB e can be constructed by taking the union of the negation of each of Conditions (C1) to (C11), and it is a rather straightforward observation that the negation of each of them can be stated as an REWB. Hence, we have that the PCP instance I has no solution if and only if $L(r) = (\Sigma_1 \times \mathcal{D})^*$. This concludes our proof in the case of multiple variables.

We now prove that we get undecidability even when using expressions with only one variable. The proof is a slight modification of the proof in multi-variable case and for completeness we present it here.

Let r be an REWB over $\Sigma[x]$.

Let $\$, \#$ be two special symbols not in Σ. Let $\Gamma = \Sigma \cup \{\$, \#\}$. Now a solution $l_1, \ldots, l_m$ of the PCP instance I can be encoded into data word $w_1 \binom{\#}{h} \mathrm{REV}(w_2)$ over $\Sigma \cup \{\$, \#\}$, where w_1, w_2 are defined as above and $\mathrm{REV}(w_2)$ is the reversal of w_2.

We then construct an REWB r over $\Gamma[x_1, \ldots, x_k]$ that accepts a data word $w = w_1 \# \mathrm{REV}(w_2)$ such that $w_1 \# w_2$ does not satisfies at least one of the Conditions (C1) to (C11) above. The REWB r is obtained by taking the union of the following.

- The negations of each (C1), (C2), (C3) which can be written in a standard regular expression without variables.

- The negation of (C4) which can be written as:

$$\Big(\Gamma^* \$ \downarrow_x . \{\Gamma^* \$ [x^=]\} \quad + \quad \Gamma^* \bigcup_{a \in \Sigma} \big(a \downarrow_x . \{\Gamma^* a [x^=]\}\big)\Big) \# \Gamma^*$$

  The negation of (C5) can be written in a similar manner.

- The negation of (C6) which can be written as:

$$\$ \downarrow_x . \{\Gamma^* \cdot \$ [x^{\neq}]\} \quad + \quad \Gamma^* \$ \downarrow_x . \{\# \cdot \Sigma^* \$ [x^{\neq}]\} \Gamma^*.$$

  The negation of (C7) can be written in a similar manner.

- The negation of (C8) which can be written as:

$$\Gamma^* \$ \downarrow_x . \left\{ \Gamma^* \# (\$ [x^{\neq}] \Sigma)^* + \Sigma^* \$ \downarrow_x . \{\Gamma * \# \Gamma * \$ [x^{\neq}]\} \Sigma^* \$ [x^=] \right\}.$$

  Note that here we use the fact that (C8) can be paraphrased as follows:

  1. For all $i \in \{1, \ldots, m-1\}$ exists $j \in \{1, \ldots, m-1\}$ such that $e_i = g_j$
  2. For all $i \in \{1, \ldots, m-1\}$ and for all $j \in \{1, \ldots, m-1\}$ if $e_i = g_j$ then $e_{i+1} = g_{j+1}$.

(Recall that by (C6) we have that $e_1 = g_1$.)

The negation of (C9) can be written in a similar manner.

- The negation of (C10) and the negation of (C11), which can be written in a straightforward manner using only one variable.

It is straightforward to see that the PCP instance I has no solution if and only if $L(r) = (\Sigma_1 \times \mathcal{D})^*$. This concludes our proof of Theorem 6.3.16. □

While restriction to simple REWBs does not make the problem decidable, the restriction to positive REWBs does: as is often the case, static analysis tasks become easier without negation.

Theorem 6.3.17. *The containment problem for positive REWBs is decidable.*

Proof. It is rather straightforward to show that any positive REWB can be converted into a register automaton without inequality [Kaminski and Tan, 2006]. The decidability of the language containment follows from the fact that the containment problem for register automata without inequality is decidable (Fact 6.1.8). □

6.4 Regular expressions with equality

Regular expressions with equality were introduced in Section 4.4 as a mechanism for defining path queries with much better complexity bounds for the query evaluation problem than register automata. Here we will redefine them in the context of data words and show that the complexity of membership and nonemptiness is much easier than in the case or register automata. Surprisingly, the universality problem is still undecidable, thus witnessing that, even strictly weaker, regular expressions with equality still retain much of the expressive power of register automata and expressions with memory or binding. Recall that the main idea of these expressions is to allow checking for (in)equality of data values at the beginning and at the end of subwords conforming to subexpressions. Next we define them formally.

Definition 6.4.1 (Expressions with equality)**.** *Let Σ be a finite alphabet. Then* regular expressions with equality (REWE) *are defined by the grammar:*

$$e \;:=\; \emptyset \mid \varepsilon \mid a \mid e+e \mid e \cdot e \mid e^+ \mid e_= \mid e_{\neq} \tag{6.2}$$

where a ranges over alphabet letters. The language $L(e)$ of data words denoted by a regular expression with equality e is defined as follows.

- $L(\emptyset) = \emptyset$.
- $L(\varepsilon) = \{\varepsilon\}$.

- $L(a) = \{\binom{a}{d} \mid d \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1 \text{ and each } w_i \in L(e)\}$.
- $L(e_=) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid d_1 = d_n\}$.
- $L(e_\neq) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid d_1 \neq d_n\}$.

Without any syntactic restrictions, there may be "pathological" expressions that, while formally defining the empty language, should nonetheless be excluded as really not making sense. For example, $\varepsilon_=$ is formally an expression, and so is $a_\neq$, although it is clear they cannot denote any data word. We exclude them by defining well-formed expressions as follows. We say that the usual regular expression e reduces to ε (respectively, to singletons) if $L(e)$ is ε or $\emptyset$ (or $|w| \leq 1$ for all $w \in L(e)$). Then we say that regular expression with equality is *well-formed* if it contains no subexpressions of the form $e_=$ or $e_\neq$, where e reduces to ε, or to singletons. From now on we will assume that all our expressions are well formed.

Note that we use $^+$ instead of $*$ for iteration. This is done for technical purposes (the ease of translation) and does not reduce expressiveness, since we can always use e^* as shorthand for $e^+ + \varepsilon$.

We now provide two examples. The expression $\Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$ denotes the language of data words that contain two different positions with the same data value. The language of data words in which the first and the last data value are different is given by $(\Sigma \cdot \Sigma^+)_\neq$.

Properties of regular expressions with equality

Connection with other languages We have already shown that, when considered over data paths, regular expressions with equality are strictly weaker than register automata. It is therefore straightforward to see that this transfers to the context of data words.

Proposition 6.4.2. *Regular expressions with equality are strictly weaker than regular expressions with memory or regular expressions with binding.*

As mentioned above, we proved this result in the case of data paths in Proposition 4.4.2. It is straightforward to adapt this proof to work for data words as well. In particular, the translation of regular expressions with memory into register automata is done by an easy inductive construction. On the other hand, to show that REWEs are strictly weaker, we can prove that they can not define the language of $(a \downarrow x) \cdot (a[x^\neq])^*$ in the same way as in the proof of Proposition 4.4.2. The only adjustment that has to be made is to redefine weak register automata over data words, much in the same manner as we have done when defining register data word automata in Section 6.1.

Closure properties As immediately follows from their definition, languages denoted by regular expressions with equality are closed under union, concatenation, and Kleene star. Also, it is straightforward to see that they are closed under automorphisms. However:

Proposition 6.4.3. *Languages recognized by regular expressions with equality are not closed under intersection and complement.*

Proof. Observe first that the expression $\Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$ defines a language of data words containing two positions with the same data value. The complement of this language is the set of all data words where all data values are different, which is not recognizable by register automata [Kaminski and Francez, 1994]. By Proposition 6.4.2 this implies that regular expressions with memory are not closed under complement.

To see that they are not closed under intersection we first show that the language

$$L = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2, d_1 \neq d_3 \text{ and } d_2 \neq d_3 \right\}$$

is not recognizable by any regular expression with equality. To prove this we simply try out all possible combinations of expressions that use at most three concatenated occurrences of a. Note that we can eliminate any expression with more that three as, or one that uses * (since this results in arbitrary long words), or union (since every member of the union would have to define words from this language and since we do not use constants we cannot just split the language into two or more parts). Also, no $=$ can occur in our expression (for subexpressions of length at least 2). This reduces the number of potential expressions to denote the language to finitely many possibilities, and we simply try them all.

Now observe that the expression $e_1 = ((a \cdot a)_{\neq} \cdot a)_{\neq}$ defines the language

$$L_1 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2 \text{ and } d_1 \neq d_3 \right\}.$$

Similarly $e_2 = a \cdot (a \cdot a)_{\neq}$ defines

$$L_2 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_2 \neq d_3 \right\}.$$

Note that $L = L_1 \cap L_2$, so if regular expressions with equality were closed under intersection they would also have been able to define the language L. □

Nonemptiness and membership To obtain fast membership and nonemptiness testing algorithms for expressions with equality, we first show how to reduce them to pushdown automata when only finite alphabets are involved.

Assume that we have a finite set D of data values. We now inductively construct PDAs $P_{e,D}$ for all regular expressions with equality e. The words recognized by these automata will be precisely the words from $L(e)$ whose data values come from D.

We construct these PDAs so that they accept by final state and furthermore have the property that only transitions of the kind $(q_0, \binom{a}{d}, X, \alpha, q)$ leave the initial state (that is any transition leaving the initial state will consume a letter) and every transition entering a final state will consume a letter. We will maintain these properties throughout the inductive construction.

It is quite clear how to construct the automata for $e = \varepsilon, e = \emptyset$ and $e = a$. For $e_1 + e_2, e_1 \cdot e_2$ and e_1^+ we use standard constructions, while for $e = (e_1)_=$, or $e = (e_1)_{\neq}$ we push the first data value on the stack, mark it by a new stack symbol and then proceed with the run of the automaton for e_1 which exists by the induction hypothesis. Every time we enter a final state of that automaton we simply empty the stack until we reach the first data value (here we use the new stack symbol) and compare it for equality or inequality with the last data value of the input word. The additional assumptions are here to assure that the construction works correctly.

Lemma 6.4.4. *The language of words accepted by each PDA $P_{e,D}$ is equal to the set of data words in $L(e)$ whose data values come from D. Moreover, the PDA $P_{e,D}$ has at most $O(|e|)$ states and $O(|e| \times (|D|^2 + |e|))$ transitions, and can be constructed in polynomial time.*

Proof. We will assume that we do not use expressions $e = \varepsilon$ and $e = \emptyset$ to avoid some technical problems. Note that this is not a problem since we can always detect the presence of these expressions in the language in linear time and code them into our automata by hand.

Assume now that we are given a well-formed regular expression with equality e (with no subexpressions of the form ε and $\emptyset$) over the alphabet Σ and a finite set of data values D. We construct, by induction on e, a PDA $P_{e,D}$ over the alphabet $\Sigma \times D$ such that:

- $w = \binom{a_1}{d_1} \dots \binom{a_n}{d_n}$ is accepted by $P_{e,D}$ if and only if $w \in L(e)$ and $d_1, \dots, d_n \in D$.
- There are no ε-transitions leaving the initial state (that is every transition from the initial state will consume a symbol).
- There is no ε-transition entering a final state.

We note that our PDAs will accept by final state and use start stack symbol.

- If $e = a$, with $a \in \Sigma$ we define $P_{e,D} = (Q, q_0, \Sigma', \Gamma, Z_0, F, \delta)$, where:
  - $Q = \{q_0, q_1\}$,
  - $F = \{q_1\}$,
  - $\Sigma' = \Sigma \times D$,
  - $\Gamma = D \cup \{Z_0\}$, and
  - $\delta(q_0, \binom{a}{d}, Z_0) = \{(q_1, \varepsilon)\}$, for every $d \in D$.

  It is straightforward to check that $P_{e,D}$ has the desired properties.

- Cases $e = e_1 + e_2$ and $e = e_1 \cdot e_2$ and $e = e_1^+$ are straightforward and are executed in a standard way using the inductive assumption to avoid ε-transitions from initial state and to final states.

- If $e = (e_1)_=$ then let $P_{e_1,D} = \{Q, q_0, \Sigma', \Gamma, Z_0, F, \delta\}$ be the PDA for e_1 and D which exists by the inductive hypothesis.

  We define $P_{e,D} = (Q', q_0, \Sigma', \Gamma', Z_0', F', \delta')$, where:

  - $Q' = Q \cup \{q', q'', q_f, q_f', q_f''\}$,
  - $F' = \{q_f\}$,
  - $\Gamma' = \Gamma \cup \{X_0\}$, where X_0 is a new stack symbol and
  - To δ' we add all the transition from δ, plus

    1. For every $(q_0, \binom{a}{d}, Z_0) \to (q_1, \alpha)$ in δ we add the transitions:
       (a) $(q_0, \binom{a}{d}, Z_0') \to (q', dZ_0')$,
       (b) $(q', \varepsilon, d) \to (q'', X_0 d)$,
       (c) $(q'', \varepsilon, X_0) \to (q'', Z_0 X_0)$, and
       (d) $(q'', \varepsilon, Z_0) \to (q_1, \alpha)$ to δ'.
    2. For every $(q_j', \binom{a}{d}, X) \to (q_j, \alpha)$ in δ, with $q_j \in F$ we add:
       (a) $(q_j', \varepsilon, X) \to (q_f', \alpha)$,
       (b) $(q_f', \varepsilon, Y) \to (q_f', \varepsilon)$, for every $Y \in \Gamma$,
       (c) $(q_f', \varepsilon, X_0) \to (q_f'', \varepsilon)$, and
       (d) $(q_f'', \binom{a}{d}, d) \to (q_f, \varepsilon)$ to δ'.

  Note first that q_1 in the first item of transitions added to δ' will never be a final state and that q_j' in the second item will never be the initial state. This simply follows from the assumption that our expressions are well-formed. Furthermore it is easy to see that no ε-transitions leave the initial state or enter a final state in our automaton.

  Next we show that the constructed automaton recognizes the language $L(e)$ restricted to data values in D. To see this note that the first block of newly added transitions simply pushes the first data value onto the stack, covers it with the new stack symbol X_0, and then proceeds as $P_{e_1,D}$ would right until the point when $P_{e_1,D}$ enters a final state. At this point $P_{e,D}$ starts to empty the stack until it sees the new symbol X_0. After popping this symbol we know that the first data value is written below it, so we compare it with the current data value for equality. If they are equal we proceed to the final state and accept (provided we have reached the end of the word).

Note that this proves that every word accepted by $P_{e,D}$ is a word accepted by $P_{e_1,D}$ that has equal first and last data value and is thus in $L(e)$ by the inductive hypothesis. The converse follows easily from this same observation and the induction hypothesis.

Note also that we can not accept any word that does not use the first transition that stores the first data value onto the stack simply because we will not have it on the stack (below X_0) when we want to proceed to the final state.

- If $e = (e_1)_{\neq}$ then let $P_{e,D}$ will be the same as for $(e_1)_{=}$, except that 2(d) changes to $(q''_f, \binom{a}{d}, d') \to (q_f, \varepsilon)$, for all $d' \neq d$ in D. The proof that this is correct is identical as in that case.

Note that the size of the stack alphabet is at most $|D| + 2|e|$, since we have to add a new stack symbol for every $=, \neq$ that appears in e (as well as the new initial stack symbol).

To see that the automaton is linear in the length of expression note that we only add new states when constructing automaton for $(e_1)_{=}, (e_1)_{\neq}$ and $e_1 + e_2$. In each case we add only a fixed number of states (five in the first two cases and one in the last).

To count the number of transitions observe that we add at most $|D|^2 + |D| + |e|$ transitions between any two states when we construct the automaton for $(e_1)_{\neq}$ (all other cases have $|D|$, or $|e|$ transitions or less). Thus we have at most $O(|e| \times (|D|^2 + |e|))$ transitions in our automaton. □

From this and Lemma 6.1.3 it is easy to obtain the following.

Theorem 6.4.5. *The nonemptiness problem for regular expressions with equality is in* PTIME.

To see this, take an arbitrary expression with equality e and convert it to a n-register data word automaton $\mathcal{A}$ that recognizes the same language. From the translation, we know that n will be at most the number of times $=$ and $\neq$ appear in e. Now do the construction from Lemma 6.4.4 for e and $D = \{0, 1, \ldots, n+1\}$ to obtain a PDA $P_{e,D}$. Proposition 6.4.2 and Lemma 6.1.3 now imply that checking if $L(e) \neq \emptyset$ is equivalent to checking $P_{e,D}$ for nonemptiness. Since this automaton is of polynomial size, we can check it for nonemptiness in PTIME thus obtaining the desired result.

Proposition 6.4.6. *The membership problem for regular expressions with equality is in* PTIME.

As in the proof of Theorem 6.4.5, we construct a PDA $P_{e,D}$ for e and $D = \{0, 1, \ldots, n\}$, where n is the length of the input word w. By invariance under automorphisms we can assume that data values in w come from the set D. Next we simply check that the word is accepted by $P_{e,D}$ and since this can be done in PTIME we get the desired result. The correctness of this algorithm follows from Lemma 6.4.4.

PDAs vs NFAs It is natural to ask whether NFAs could not have been used instead of pushdown automata. The answer is that they can be used to capture languages of data words described by regular expressions with equality over a finite set of data values, but the cost is necessarily exponential, and hence we cannot possible use them to derive Theorem 6.4.5. That is, we can first show:

Proposition 6.4.7. *For every regular expression with equality e over the alphabet Σ and a finite set D of data values there exists an NFA $\mathcal{A}_{e,D}$, of the size exponential in $|e|$, recognizing precisely those data words from $L(e)$ that use data values from D.*

Proof. We prove this by structural induction on regular expressions with equality. All of the standard cases are carried out as usual. Thus we only have to describe the construction for subexpressions of the form $e_{=}$ and $e_{\neq}$. In both cases by the induction hypothesis we know that there is an NFA $\mathcal{A}_{e,D}$ recognizing words in $L(e)$ with data values from D. The automaton for $\mathcal{A}_{e_{\neq},D}$ (and likewise for $\mathcal{A}_{e_{=},D}$) will consist of $|D|$ disjoint copies of $\mathcal{A}_{e,D}$, each designated to remember the first data value read when processing the input. According to this, whenever our automaton would enter a final state we test that the current data value is different (or the same) to the one corresponding to this copy of the original automaton. This is done in a manner analogous to the one used in the proof of Proposition 6.4.4. □

However, the exponential lower bound is the best we can do in the general case. To see this, we define a sequence of regular expressions with memory $\{e_n\}_{n\in\mathbb{N}}$, over the alphabet $\Sigma = \{a\}$, and each of length linear in n. We then show that for $D = \{0,1\}$ every regular expression over the alphabet $\Sigma \times D$ recognizing precisely those data words from $L(e_n)$ with data values in D has length exponential in $|e_n|$.

To prove this we will use the following theorem for proving lower bounds of NFAs [Glaister and Shallit, 1996]. Let $L \subseteq \Sigma^*$ be a regular language and suppose there exists a set $P = \{(x_i, y_i) : 1 \leq i \leq n\}$ of pairs such that:

1. $x_i \cdot y_i \in L$, for every $i = 1, \ldots n$, and
2. $x_i \cdot y_j \notin L$, for $1 \leq i, j \leq n$ and $i \neq j$.

Then any NFA accepting L has at least n states.

Thus to prove our claim it suffices to find such a set of size exponential in the length of e_n. Next we define the expressions e_n inductively as follows:

- $e_1 = (a \cdot a)_{=}$,
- $e_{n+1} = (a \cdot e_n \cdot a)_{=}$.

It is easy to check that $L(e_n) = \{w \cdot w^{-1} : w \in (\Sigma \times \{0,1\})^n\}$, where w^{-1} denotes the reverse of w.

Now let $w_1, \ldots w_{2^n}$ be a list of all the elements in $(\Sigma \times \{0,1\})^n$ in arbitrary order. We define the pairs in P as follows:

- $x_i = w_i$,
- $y_i = (w_i)^{-1}$.

Since these pairs satisfy the above assumptions 1) and 2), we conclude, using the result of [Glaister and Shallit, 1996], that any NFA recognizing $L(e_n)$ has at least $O(2^{|e_n|})$ states, so no regular expression describing it can be of length polynomial in $|e_n|$.

Containment and universality Surprisingly one can show that even this relatively weak class of expressions still retains enough power to code PCP when its universality problem is considered. Form this also follows that language containment is undecidable.

Proposition 6.4.8. *Universality and containment are undecidable for regular expressions with equality.*

Proof. The proof is basically identical to the proof of Theorem 6.3.16. One only has to notice that each of the REWBs expressing negation of conditions (C1) to (C11) in that proof can easily be replaced by an equivalent expression with equality.

For example, the negation of (C4) can be written as:

$$\left(\Gamma^*\$(\Gamma^*\$)_= \quad + \quad \Gamma^* \bigcup_{a\in\Sigma} \left(a(\Gamma^* a)_=\right)\right)\#\Gamma^*$$

Similarly, the negation of (C6) is expressed by:

$$\$(\Gamma^* \cdot \$)_{\neq} \quad + \quad \Gamma^*\$(\# \cdot \Sigma^*\$)_{\neq}\Gamma^*.$$

The negation of other expressions can be expressed in an analogous manner. When examining the query containment problem for RQDs in Chapter 10 we will present the proof in full detail. □

6.5 Variable automata

Final data word defining mechanism we will consider is the one of Variable automata. Recall that we already studied variable automata over data paths in Section 4.5. Here we will show that they can also be defined over data words, thus eliminating the need to have a separate set of word states and data states, as one does when working with data paths.

Although most of the results presented in this section will easily follow from [Grumberg et al., 2010a], where variable automata were first introduced as a means to define languages over an infinite alphabet, we include them here to have a complete picture of currently available data word formalisms.

We begin by defining variable automata over data words.

Definition 6.5.1. *Let Σ be a finite alphabet and $\mathcal{D}$ an infinite domain of data values. We will also assume that we have a countable set V of variables. A* variable finite automaton *(or VFA for short) over $\Sigma \times \mathcal{D}$ is a pair $\mathcal{A} = (\Gamma, A)$, where A is an NFA over the alphabet $\Sigma \times \Gamma$, and $\Gamma = C \cup X \cup \{\star\}$ such that:*

- *$C \subseteq \mathcal{D}$ is a finite set of data values called* constants
- *$X \subseteq V$ is a finite set of* bound variables*, and*
- *$\star$ is a symbol for the* free variable*.*

Next we define when a VFA accepts a data word $w = w_1 w_2 \dots w_n \in (\Sigma \times \mathcal{D})^*$. For each letter $u = \binom{a}{d}$ in $\Sigma \times \mathcal{D}$, we let $\lambda(u) = a$ (label projection) and $\delta(u) = d$ (data projection).

Let $v = v_1 v_2 \dots v_n \in (\Sigma \times \Gamma)^*$ be a word accepted by A. We will say that v is a *witnessing pattern* for w (or that w is a *legal instance* of v) if the following holds:

1. $\lambda(v_i) = \lambda(w_i)$, for $i = 1, \dots, n$,
2. $\delta(v_i) = \delta(w_i)$ whenever $\delta(v_i) \in C$,
3. if $\delta(v_i), \delta(v_j) \in X$, then $\delta(w_i), \delta(w_j) \notin C$ and $\delta(w_i) = \delta(w_j)$ iff $\delta(v_i) = \delta(v_j)$,
4. if $\delta(v_i) = \star$ and $\delta(v_j) \neq \star$, then $\delta(w_i) \neq \delta(w_j)$.

Intuitively the definition states that in a legal instance constants and finite alphabet part will remain unchanged (conditions 1 and 2), while every bound variable is assigned with the same *unique* data value from $\mathcal{D} - C$ (condition 3) and every occurrence of the free variable $\star$ is freely assigned any data value from $\mathcal{D} - C$ that is not assigned to any of the bound variables (condition 4). Note that the condition 4 is a lot stronger that saying that $\star$ is just a wild card.

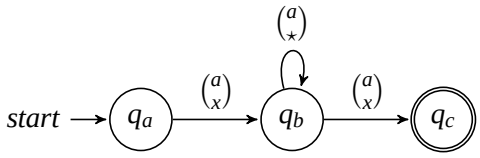
We now define the *language of $\mathcal{A}$*, or simply $L(\mathcal{A})$ for short, as the set of all data words w for which there exists a witnessing pattern $v \in L(A)$. That is a word is accepted by $\mathcal{A}$ if there is a witnessing pattern for it that is accepted by the underlying NFA A.

Note that it is straightforward to define regular expressions for VFAs that will simply inherit the associated semantics.

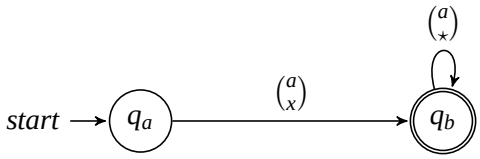
Remark 7. *Note that VFAs when defined over data words differ slightly from the ones defined over data paths. The reason for this is that over data words there is no asymmetry when defining concatenation, as in the case for data paths. Therefore, we no longer need two separate sets of states, so the automaton itself can be represented by the runs of a single NFA A as in the definition above. However, the idea of guessing values in advance is identical in both approaches and it is not difficult to see how one can go from one setting to the other, much like in Section 3.1.*

Example 6.5.2. *Here we give a few examples of languages accepted by VFAs.*

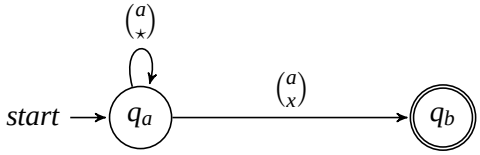
1. *The language where the first data value is equal to the last and all other values are different from them (but can be equal among themselves).*



2. *The language where the first data value is different from all other data values.*



3. *The language where the last data value differs from all other data values.*



Note that the last example is not expressible by register automata [Kaminski and Francez, 1994].

It was shown in [Grumberg et al., 2010b] that the language $L = \{\binom{a}{d_1}\binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_2}\dots\binom{a}{d_k}\binom{a}{d_k} \mid k \geq 1\}$ is not expressible by VFAs. (Note that there VFA were disregarding finite labels, but this already implies our claim.) However, it is straightforward to show that it is expressible by a regular expression with equality $((aa)_=)^+$. Thus, we obtain:

Proposition 6.5.3. *VFAs are incomparable in terms of expressive power with register automata, regular expressions with memory, regular expressions with binding and regular expressions with equality.*

Closure and decision problems for VFAs

As already mentioned, most of the results below readily follow from [Grumberg et al., 2010a, Grumberg et al., 2010b]. For the sake of completeness we also include them here.

Closure properties When it comes to closure properties VFAs behave in a similar manner to register automata and regular expressions with memory. Namely we have the following.

Fact 6.5.4 ([Grumberg et al., 2010a, Grumberg et al., 2010b])**.** *1. The set of languages recognized by variable automata is closed under union, intersection, concatenation and Kleene star.*

2. Languages recognized by variable automata are not closed under complement.

Although the proofs presented in [Grumberg et al., 2010b] do not consider data words it is straightforward to see that an analogous construction can be carried out in this setting.

Decision problems The somewhat unnatural behaviour of VFAs is exhibited in terms of decision problems. In particular, one can show that nonemptiness amounts to no more than checking nonemptiness of the underlying NFA, thus bringing the complexity down to NLOGSPACE-complete , unlike in the case of e.g. register automata. On the other hand, membership is significantly harder and the complexity here jumps to NP-complete, since one can easily code hamiltonicity using variables (see Theorem 5 in [Grumberg et al., 2010b]). Therefore we can conclude that the use of variables leads to unusual behaviour, as one usually exprects the membership problem to be easier that nonemptiness.

Fact 6.5.5 ([Grumberg et al., 2010a, Grumberg et al., 2010b])**.** *1. The nonemptiness problem for VFAs is* NLOGSPACE*-complete.*

2. The membership problem for VFAs is NP*-complete.*

Unsurprisingly, one can show that containment and universality are also undecidable by modifying the proof in [Neven et al., 2004] to the context of VFAs.

Fact 6.5.6 ([Grumberg et al., 2010b])**.** *Both containment and universality problems are undecidable for VFAs.*

To get a decidable subcase of the language containment problem (and thus also universality), we turn to restriction based on *deterministic variable automata – DVFAs.* These are the VFAs with the property that for every word in their language there is only one run accepting it. Note that these are not the same as the ones whose underlying NFA is deterministic. It can then be shown that:

Fact 6.5.7 ([Grumberg et al., 2010b])**.** *The containment problem for* deterministic *VFAs is in* coNP.

Although testing if a VFA is deterministic can be done in NL, problem of determinizing VFAs is undecidable [Grumberg et al., 2010b]. There is however a nice class of determinizable VFAs – the ones with no free variable mentioned in the underlying NFA. It is easy to see that this fragments corresponds to regular expressions with backreferencing [Aho, 1990], which are, in essence, grep specifications from the Unix systems.

6.6 Summary of language theoretic properties

When main computational tasks are concerned we see that complexity of the nonemptiness problem basically matches the bounds on combined complexity of query evaluation, apart from the case of variable automata and expressions with binding. This fact, in conjunction with the query evaluation algorithms presented in Chapter 4 which rely on checking NFA nonemptiness, might lead to a conclusion that the two problems are closely related. However, it is important to note that this is not the case. Indeed, the mentioned evaluation algorithms simply use the fact that all possible paths in a graph, together with the query, can be coded by an exponential size NFA. This further exemplifies the two degrees of separation in path queries, where paths are selected beforehand, and then their labels are checked for membership in the language theoretic formalism defining them. The nonemptiness problem on the other hand, reasons about the query itself, not taking a particular graph into the account. It can, for example, be the case that language of an expression with memory is nonempty, while the answer to the corresponding RQM produces no output on some particular graph. Indeed, there are graphs where no path query will have a nonempty answer. The difference becomes even more apparent when REWBs are considered, since here the nonemptiness problem enjoys lower complexity that that for evaluation of the associated class of graph queries.

We also studied membership, complexity of which ranges from PTIME to NP, as well as universality and query containment. The latter two were shown to be undecidable for all of the formalisms studied in this chapter, however we did isolate several decidable fragments. We will return to this question later on in Chapter 10 where finding decidable fragments becomes crucial for the static analysis aspects of graph query languages.

The summary of the complexity bounds for nonemptiness, membership and containment/universality is presented in Table 6.1.

	RA	REM	REWB	REWE	VFA
nonemptiness	PSPACE-c	PSPACE-c	NP-c	PTIME	NLOGSPACE-c
membership	NP-c	NP-c	in NP	PTIME	NP-c
containment	undecidable	undecidable	undecidable	undecidable	undecidable
universality	undecidable	undecidable	undecidable	undecidable	undecidable

Table 6.1: Complexity of main decision problems

As is common in language theory, we also studied basic closure properties of our languages. A summary of the results is given in Table 6.2. We can see that while all of the formalisms are closed under union, concatenation and Kleene star, none is closed under complementation. The main reason for this lies in the fact that closure under complement (together with the ability to define one of the most basic languages where two data values are equal) would yield a high query evaluation bound (see Theorem 3.2.1), making the formalism unsuitable for querying graphs. We also studied closure under intersection, and while most languages do enjoy this property (due to a fact that one can carry out the standard NFA product construction), for the case of REWBs and REWEs we can show that this is no longer true.

	RA	REM	REWB	REWE	VFA
union	+	+	+	+	+
intersection	+	+	−	−	+
concatenation	+	+	+	+	+
Kleene star	+	+	+	+	+
complement	−	−	−	−	−

Table 6.2: Closure properties of data word defining formalisms

Lastly, we also studied how the five classes of languages compare one to another. While regular expressions with memory were originally introduced as an expression analogue of register automata, here we also showed that they subsume expressions with binding as well as expressions with equality. Moreover, it is readily checked that the language shown not expressible by regular expressions with equality in Proposition 6.4.2 is captured by REWBs, giving us another proper inclusion. VFAs, on the other hand, are orthogonal to all the other formalisms studied in this chapter, as they can express properties out of the reach of register automata, while failing to capture even REWEs. We thus obtain:

Theorem 6.6.1. *The following relations hold, where $\subsetneq$ denotes that every language defined by formalism on the left is definable by the formalism on the right, but not vice versa.*

- *REWEs $\subsetneq$ REWBs $\subsetneq$ REMs $=$ register automata.*
- *VFAs are incomparable in terms of expressive power with REWEs, REWBs, REMs and register automata.*

Part II

Graph languages and beyond

Chapter 7

Graph XPath

In Chapter 4 we have seen several languages for describing properties of paths in data graphs, but for some applications paths alone are no longer sufficient. Consider again the database from Figure 2.3. Here one might redefine the notion of Bacon number in such a way that each collaboration witnessing it has to go through movies; documentaries will not suffice any more. Such a query lies outside of reach of any path language, since at each point of the path one has to check if the actors co-starred in a movie. Note that even conjunctive path queries can not express this property, since the test has to be carried out for an arbitrary number of steps. Therefore, in order to define such queries one needs languages that allow for patterns that are no longer only paths, but allow testing if every point along a path has some property. Another issue with path languages is that they are inherently binary. But for instance, if we want to find people with a finite Bacon number, we are asking a unary query. Then why not allow languages to return only the source of a path or a pattern that conforms to the query?

Note that the well studied XML language XPath has the ability to do both of these two things. It is also important to observe that the goal of XPath seems very similar to the goal of many queries in graph databases: it describes properties of paths and patterns, taking into account both their purely navigational aspects as well as the data that is found in XML documents. The popularity of XPath is largely due to several factors:

- it defines many properties of paths and patterns that are relevant for navigational queries;
- it achieves expressiveness that relates naturally to yardstick languages for databases (such as first-order logic, its fragments, or extensions with some form of recursion); and
- it has good computational properties over XML, notably tractable combined complexity for many fragments and even linear-time complexity for some of them.

A natural question then is to see if main ingredients that made XPath successful in the context of XML can be applied on graphs. In what follows we will address this issue and show

that when applied to graphs XPath-like languages define an efficient and highly expressive class of queries.

There appear to be two ways to use XPath as a graph database language. The first possibility is to essentially stick to the idea of RPQs and use XPath to describe paths between nodes, thus making it a path language. While XPath on words with data is well understood by now [Bojanczyk and Lasota, 2010, Figueira, 2010b], this idea has several drawback. First of all, XPath is intrinsically a graph (originally tree) language, and even when it is used to reason about data words the semantics relies on defining patterns (see e.g. [Figueira and Segoufin, 2009], or Part I in [Figueira, 2010b]) in the same way as on trees. Indeed, when used over data words XPath simply treats them as trees and is thus not a true path language. Another reason not to study XPath as a path language is that even the more general graph approach already yields very efficient query evaluation algorithms (combined complexity is always PTIME and for some fragments even linear). It therefore makes little sense to sacrifice expressive power for no palpable gain in efficiency while at the same time making the language somewhat artificial.

A different approach is to apply XPath queries to the entire graph database, rather than use them to define sets of allowed paths. This is the approach we pursue. To a limited extent it was tried before. On the practical side, XPath-like languages have been used to query graph data (e.g., [Cassidy, 2003, Gremlin, 2013]), without any analysis of their expressiveness and complexity, however. On the theoretical side, several papers investigated XPath-like languages from the modal perspective, dropping the assumption that they are evaluated on trees [Alechina et al., 2003, Marx, 2003], but most notably in [Fletcher et al., 2011] the authors consider an algebra of binary relations which is the basis of our navigational language. It is important to note that none of these approaches considered data values, thus making them suited only to ask queries about topology of the graph and not about the interplay this topology has with the stored data.

Thus, our goal is to investigate how XPath-languages can be used to query graph databases. In particular, we want to understand both the navigational querying power of such languages, and their ability to handle navigation and data together in graph databases. In this investigation, we can take advantage of the vast existing XML literature on algorithmic and language-theoretic aspects of XPath.

We use several versions of XPath-like languages for graph databases, all of them collectively named GXPath. The core language is denoted by $\mathsf{GXPath}_{\mathrm{core}}$ and is basically an adaptation of Core XPath 2.0 [ten Cate and Marx, 2007, Xpath 2.0, 2010] for graphs. The analogue of regular XPath, allowing arbitrary transitive closure, is called $\mathsf{GXPath}_{\mathrm{reg}}$. Like XPath (or closely related logics such as PDL and CTL*), all versions of GXPath have node tests and path formulae, and as the basic axes they use letters from the alphabet labelling graph edges. For instance, $a^* \cdot (b^-)^*$ finds pairs of nodes connected by a path that starts with a-edges in the

forward direction, followed by b-edges in the backward direction. Formulae may also include node tests: for instance, $a^*[c] \cdot (b^-)^*$ modifies the above expression by requiring that the node where the a-labels switch to b-labels also has an outgoing c-edge. And crucially, node tests can refer to data values and have XPath-like conditions over them. For instance, the expression $a^*[=5] \cdot (b^-)^*$ checks if the data value in that intermediate node is 5, and $a^*[\langle a = b \rangle] \cdot (b^-)^*$ checks if that node has two outgoing edges, labelled a and b, to nodes that store the same data value.

We first study the complexity of various fragments of GXPath. As it turns out, *all* GXPath fragments inherit nice properties from XPath on trees due to the 'modal' nature of the language: the combined complexity is always polynomial. Even more, it is always a low-degree polynomial. In fact, the query complexity is linear for all the fragments we consider. The data complexity is not worse than cubic for navigational $\mathsf{GXPath}_{\mathrm{reg}}$ and linear for its positive fragments. With data comparisons added, data complexity becomes cubic again. We also show that adding numerical formulas that specify length of a path connecting two nodes, although making the language exponentially more succinct [Losemann and Martens, 2012], has no effect on the complexity of query evaluation.

Following this we analyse the expressive power of the language, using the usual database yardstick of first-order logic as our reference point. It turns out that $\mathsf{GXPath}_{\mathrm{core}}$ captures precisely FO^3, first-order logic with 3 variables, like its analog (core XPath 2.0) on trees. The difference, though, is that on graphs $\mathrm{FO} \neq \mathrm{FO}^3$, but on trees the two are the same. Note that on trees there is another way of capturing FO, by means of *conditional* XPath [Marx, 2005], which adds the until-operator. We show that on graphs the analog of conditional XPath goes beyond FO. We also show how $\mathsf{GXPath}_{\mathrm{reg}}$ can be captured by a parameter-free fragment of transitive closure logic FO^*.

Since these comparisons were done without taking data values into account, we next consider FO that has the capability of comparing data values, denoted $\mathrm{FO}(\sim)$. Although we show that using standard XPath data tests falls short of capturing $\mathrm{FO}(\sim)$, when same tests as in *RQD*s are used, the result again follows.

Finally, we establish the full hierarchy of various GXPath fragments and variants and show how they can be extended with conjunction, allowing us even more expressive power with optimal efficiency.

7.1 The language and its many variants

We follow the standard way of defining XPath fragments [Bojanczyk and Parys, 2011, Calvanese et al., 2009, Figueira, 2010b, Gottlob et al., 2005, Marx, 2005, ten Cate and Marx, 2007] and introduce some variants of *graph XPath*, or GXPath, to be interpreted over graph databases.

As usual, XPath formulae are divided into *path formulae*, producing sets of pairs of nodes, and *node tests*, producing sets of nodes. Path formulae will be denoted by letters from the beginning of the Greek alphabet ($\alpha, \beta, \ldots$) and node formulae by letters from the end of the Greek alphabet ($\varphi, \psi, \ldots$).

Since we deal with data values, we need to define *data tests* permitted in our formulas. There will be three kinds of them.

1. Constant tests: For each data value $c \in \mathcal{D}$, we have two tests $=c$ and $\neq c$. The intended meaning is to test whether the data value in the current node equals to, or differs from, constant c.

   The fragment of GXPath that uses constant tests will be denoted by $\mathsf{GXPath}(\mathsf{c})$.

2. Equality/inequality tests: These are typical XPath (in)equality tests of the form $\langle \alpha = \beta \rangle$ and $\langle \alpha \neq \beta \rangle$, where α and β are path expressions. The intended meaning is to check for the existence of two paths, one satisfying α and the other satisfying β, which end with equal (resp., different) data values.

   The appropriate fragment will be denoted by $\mathsf{GXPath}(\mathsf{eq})$. If we have both constant tests and equality tests, we denote resulting fragments by $\mathsf{GXPath}(\mathsf{c}, \mathsf{eq})$.

3. Subexpression tests: These are used to test if a path or a subpath starts and ends with the same or different data value.

   The fragment in question is obtained by adding $\alpha_{=}$ and $\alpha_{\neq}$ to path expressions of our language. These tests will be needed to provide a logical kernel for GXPath.

   The corresponding fragment is denoted $\mathsf{GXPath}(\sim)$.

Next we define expressions of GXPath. As already mentioned, we look at *core* and *regular* versions of XPath. They both have node and path expressions. Node expressions in all fragments are given by the grammar:

$$\varphi, \psi := \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle$$

where α is a path expression.

The path formulae of the two flavours of GXPath are given below. In both cases a ranges over Σ.

Path expressions of *Regular graph XPath*, denoted by $\mathsf{GXPath}_{\mathrm{reg}}$, are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^{-} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha} \mid \alpha^{*}$$

Path expressions of *Core graph XPath* denoted by $\mathsf{GXPath}_{\mathrm{core}}$ are given by:

$$\alpha, \beta := \varepsilon \mid \_ \mid a \mid a^{-} \mid a^{*} \mid a^{-*} \mid [\varphi] \mid \alpha \cdot \beta \mid \alpha \cup \beta \mid \overline{\alpha}$$

We call this fragment "Core graph XPath", since it is natural to view edge labels (and their reverse) in data graphs as the single-step axes of the usual XPath on trees. For instance, a and a^- could be similar to "child" and "parent". Thus, in our core fragment, we only allow transitive closure over navigational single-step axes, as is done in Core XPath on trees. Note that we did not explicitly define the counterpart of node label tests in GXPath node expressions to avoid notational clutter, but all the results remain true if we add them.

Finally, we consider another feature that was recently proposed in the context of navigational languages on graphs (such as in SPARQL 1.1 [Harris and Seaborne, 2013]), namely counters. The idea is to extend all grammars defining path formulae with new path expressions

$$\alpha^{n,m}$$

for $n,m \in \mathbb{N}$ and $n < m$. Informally, this means that we have a path that consists of some k chunks, each satisfying α, with $n \leq k \leq m$.

When counting is present in the language, we denote it by #GXPath, e.g., $\#\mathsf{GXPath}_{\text{core}}$.

Given these path and node formulae, we can combine $\mathsf{GXPath}_{\text{core}}$ and $\mathsf{GXPath}_{\text{reg}}$ with different flavours of data tests or counting, starting with purely navigational fragments (neither c, eq, nor $\sim$ tests are allowed) and up to fragments allowing any combination of such tests. For example, $\#\mathsf{GXPath}_{\text{reg}}(\mathsf{c},\mathsf{eq})$ is defined by mutual recursion as follows:

$$\alpha,\beta := \varepsilon \mid \_ \mid a \mid a^- \mid [\varphi] \mid \alpha\cdot\beta \mid \alpha\cup\beta \mid \overline{\alpha} \mid \alpha^* \mid \alpha^{n,m}$$
$$\varphi,\psi := \neg\varphi \mid \varphi\wedge\psi \mid \langle\alpha\rangle \mid {=}c \mid {\neq}c \mid \langle\alpha=\beta\rangle \mid \langle\alpha\neq\beta\rangle$$

with c ranging over constants, while $\mathsf{GXPath}_{\text{reg}}(\sim)$ is given by:

$$\alpha,\beta := \varepsilon \mid \_ \mid a \mid a^- \mid [\varphi] \mid \alpha\cdot\beta \mid \alpha\cup\beta \mid \overline{\alpha} \mid \alpha^* \mid \alpha_{=} \mid \alpha_{\neq}$$
$$\varphi,\psi := \top \mid \neg\varphi \mid \varphi\wedge\psi \mid \varphi\vee\psi \mid \langle\alpha\rangle$$

We define the semantics with respect to a data graph $G = \langle V, E, \rho\rangle$. The semantics $[\![\alpha]\!]^G$ of a path expression α is a set of pairs of vertices and the semantics of a node test, $[\![\varphi]\!]^G$, is a set of vertices. The definitions are given in Figure 7.1. In that definition, by R^k we mean the k-fold composition of a binary relation R, i.e., $R\circ R\circ\ldots\circ R$, with R occurring k times.

Remark. Note that each path expression α can be transformed into a node test by the means of $\langle\alpha\rangle$ operator. In particular, we can test if a node has a b-successor by writing, for instance, $\langle b\rangle$. To reduce the clutter when using such tests in path expressions, we shall often omit the $\langle\rangle$ braces and write e.g. $a[b]$ instead of $a[\langle b\rangle]$.

Basic expressiveness results Some expressions are readily definable with those we have. For instance, Boolean operations $\alpha\cap\beta$ and $\alpha-\beta$ with the natural semantics are definable. Indeed, $\alpha-\beta$ is definable as $\overline{\overline{\alpha}\cup\beta}$, and intersection is definable with union and complement. So when necessary, we shall use intersection and set difference in path expressions.

Path expressions

$$
\begin{aligned}
[\![\varepsilon]\!]^G &= \{(v,v) \mid v \in V\} \\
[\![\_]\!]^G &= \{(v,v') \mid (v,a,v') \in E \text{ for some } a\} \\
[\![a]\!]^G &= \{(v,v') \mid (v,a,v') \in E\} \\
[\![a^-]\!]^G &= \{(v,v') \mid (v',a,v) \in E\} \\
[\![\alpha^*]\!]^G &= \text{the reflexive transitive closure of } [\![\alpha]\!]^G \\
[\![\alpha \cdot \beta]\!]^G &= [\![\alpha]\!]^G \circ [\![\beta]\!]^G \\
[\![\alpha \cup \beta]\!]^G &= [\![\alpha]\!]^G \cup [\![\beta]\!]^G \\
[\![\overline{\alpha}]\!]^G &= V \times V - [\![\alpha]\!]^G \\
[\![[\varphi]]\!]^G &= \{(v,v) \in G \mid v \in [\![\varphi]\!]^G\} \\
[\![\alpha^{n,m}]\!]^G &= \bigcup_{k=n}^{m} ([\![\alpha]\!]^G)^k \\
[\![\alpha_=]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\} \\
[\![\alpha_{\neq}]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\}
\end{aligned}
$$

Node tests

$$
\begin{aligned}
[\![\langle\alpha\rangle]\!]^G &= \pi_1([\![\alpha]\!]^G) = \{v \mid \exists v' \ (v,v') \in [\![\alpha]\!]^G\} \\
[\![\top]\!]^G &= V \\
[\![\neg\varphi]\!]^G &= V - [\![\varphi]\!]^G \\
[\![\varphi \wedge \psi]\!]^G &= [\![\varphi]\!]^G \cap [\![\psi]\!]^G \\
[\![\varphi \vee \psi]\!]^G &= [\![\varphi]\!]^G \cup [\![\psi]\!]^G \\
[\![=c]\!]^G &= \{v \in V \mid \rho(v) = c\} \\
[\![\neq c]\!]^G &= \{v \in V \mid \rho(v) \neq c\} \\
[\![\langle\alpha = \beta\rangle]\!]^G &= \{v \in V \mid \exists v', v'' \ (v,v') \in [\![\alpha]\!]^G, \ (v,v'') \in [\![\beta]\!]^G, \rho(v') = \rho(v'')\} \\
[\![\langle\alpha \neq \beta\rangle]\!]^G &= \{v \in V \mid \exists v', v'' \ (v,v') \in [\![\alpha]\!]^G, \ (v,v'') \in [\![\beta]\!]^G, \rho(v') \neq \rho(v'')\}
\end{aligned}
$$

Figure 7.1: Semantics of Graph XPath expressions with respect to $G = \langle V, E, \rho \rangle$

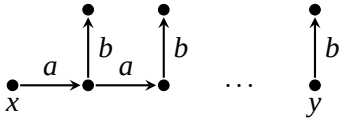
Counting expressions $\alpha^{n,m}$ are definable too: they abbreviate $\alpha\cdots\alpha\cdot(\alpha\cup\varepsilon)\cdots(\alpha\cup\varepsilon)$, where we have a concatenation of n times α and $m-n$ times $(\alpha\cup\varepsilon)$. Thus, adding counters does not influence expressivity of any of the fragments, since we always allow concatenation and union. However, counting expressions can be exponentially more succinct than their smallest equivalent regular expressions (independent of whether n and m are represented in binary or in unary) [Losemann and Martens, 2012]. We will exhibit a query evaluation algorithm with polynomial-time complexity even for such expressions with counters represented in binary.

As another observation on the expressiveness of the language, note that we can define a test $\langle\alpha = c\rangle$, with the semantics $\{v \mid \exists v'\ (v,v') \in [\![\alpha]\!]^G$ and $\rho(v') = c\}$, by using the expression $\langle\alpha[{=}c]\rangle$.

Another thing worth noting is that node expressions can be defined in terms of path operators. For example $\varphi\wedge\psi$ is defined by the expression $\langle[\varphi]\cdot[\psi]\rangle$, while $\neg\varphi$ is defined by $\langle\overline{[\varphi]}\cap\varepsilon\rangle$.

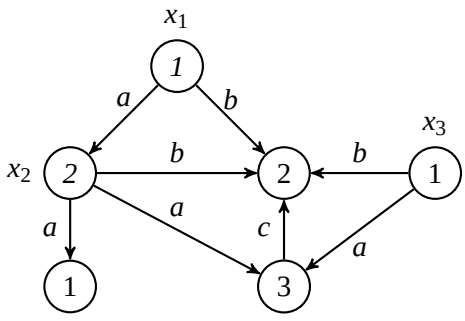
Example 7.1.1. *We next give a few examples of GXPath expressions to illustrate what sort of queries one can ask using these languages.*

1. *The expression $(a[b])^*$ will simply give us all pairs (x,y) of nodes that are connected by a path of the following form:*



   That is, x and y are connected by an a^* labelled path such that each node on the path also has an outgoing b-labelled edge. (Nodes that are different in the picture do not have to be different in the graph.)

2. *The expression $\langle aa^* \neq bc^-\rangle$ will give us all nodes x such that there are nodes y and z, reachable by aa^* and bc^- respectively, with different data values. For example in the graph given in the following image the nodes x_1 and x_2 will be selected by our query, while x_3 will not.*



3. *The expression $\langle(a[{=}5]\cdot(a[{=}5])^*)\cap\varepsilon\rangle$ will extract all the nodes x such that there is a cycle starting at x in which each edge is labelled by a and each node has the data value*

5. In particular the node x will have data value 5. Note that this example illustrates how we can define loops using GXPath.

To illustrate some more involved queries we come back to our introductory example of a movie database presented in Figure 2.3.

Example 7.1.2. *1. To find people with a finite Bacon number we simply use the query*

$$e_1 = \langle (\mathsf{cast}^- \cdot \mathsf{cast})^* [= \mathsf{Kevin\ Bacon}] \rangle.$$

Similarly as in the example with path languages, the query traverses cast edges checking for collaborations and in the end makes sure that the actor reached is Kevin Bacon. Note that this is a unary query, so we no longer have to return additional information, such as the node corresponding to Kevin Bacon, as we did when dealing with path queries.

2. *Using path negation we can also find actors who do not have a finite Bacon number. Such a query is of interest when we want to see if every actor in the database does have a Bacon number – we simply ask the query and check if the answer is nonempty. The query is given by*

$$e_2 = \langle \overline{(\mathsf{cast}^- \cdot \mathsf{cast})^* [= \mathsf{Kevin\ Bacon}]} \rangle.$$

3. *As mentioned in the introduction movie databases often allow searching through a specific genre, so for example we might want to find actors who have a finite Bacon number, but such that the collaboration is always established by co-starring in movies and not documentaries. This query is as follow:*

$$e_3 = \langle (\mathsf{cast}^- [\mathsf{type} [= \mathsf{Movie}]] \cdot \mathsf{cast})^* [= \mathsf{Kevin\ Bacon}] \rangle.$$

This expression works in a similar way as the one for finding the Bacon number, but using the nesting capabilities of GXPath it also checks that the actors appear in a movie.

4. *One might also be interested to find out if there are actors who have a finite Bacon number and the same age as Kevin Bacon. They can be retrieved using the following query:*

$$e_4 = \langle (\mathsf{age}^- (\mathsf{cast}^- \cdot \mathsf{cast})^* [= \mathsf{Kevin\ Bacon}])_= \rangle.$$

5. *As a last example we might want to check if a movie or a documentary has at least two actors starring in it. Such a query is defined by:*

$$e_5 = \langle \mathsf{cast} \neq \mathsf{cast} \rangle.$$

Here we simply check if there are two cast edges leading from the movie such that the actors names are different.

Complement and positive fragments In standard XPath dialects on trees, complementation operator is not included and one usually shows that languages are closed under negation. This is no longer true for arbitrary graphs, due to the following.

Proposition 7.1.3. *Path complementation $\overline{\alpha}$ is not definable in GXPath$_{reg}$ without complement on path expressions.*

The proof is an immediate consequence of the following observation. Given a data graph G, let $V_1, \ldots, V_m$ be sets of nodes of its (maximal) connected components (with respect to the edge relation $\bigcup_{a\in\Sigma} E_a$). Then a simple induction on the structure of the expressions of GXPath$_{\text{reg}}$ without complement on path expressions shows that for each expression α, we have $[\![\alpha]\!]^G \subseteq \bigcup_{i\leq m} V_i \times V_i$. However, path complementation $\overline{\alpha}$ clearly violates this property.

In what follows, we consider fragments of our languages that restrict complementation and negation. There are two kinds of them, the first corresponding to the well-studied notion of positive XPath.

- The *positive fragments* are obtained by removing $\neg\varphi$ and $\overline{\alpha}$ from the definitions of node and path formulae. We use the superscript pos to denote them, i.e., we write GXPath$^{\text{pos}}_{\text{core}}$ and GXPath$^{\text{pos}}_{\text{reg}}$.
- The *path-positive fragments* are obtained by removing $\overline{\alpha}$ from the definitions of path formulae, but keeping $\neg\varphi$ in the definitions of node formulae. We use the superscript path-pos to denote them, i.e., we write GXPath$^{\text{path-pos}}_{\text{core}}$ and GXPath$^{\text{path-pos}}_{\text{reg}}$.

7.2 Query evaluation

In this section we investigate the complexity of querying graph databases using variants of GXPath. We consider two problems. One is QUERY EVALUATION, which is essentially model checking: we have a graph database, a query (i.e., a path expression), and a pair of nodes, and we want to check if the pair of nodes is in the query result. That is, we deal with the following decision problem.

PROBLEM:	QUERY EVALUATION
INPUT:	A graph $G = (V, E)$,
	a path expression α, nodes $v, v' \in V$.
QUESTION:	Is $(v, v') \in [\![\alpha]\!]^G$?

The second problem we consider is QUERY COMPUTATION, which actually computes the result of a query and outputs it. Normally, when one deals with path expressions, one fixes a

so-called *context node* v and looks for all nodes v' such that (v,v') satisfies the expression. We deal with a slightly more general version here, where there can be a set of context nodes instead of just a single one.

PROBLEM:	QUERY COMPUTATION
INPUT:	A graph $G=(V,E)$, a path expression α, and a set of nodes $S \subseteq V$.
OUTPUT:	All $v' \in V$ such that there exists a $v \in S$ with $(v,v') \in [\![\alpha]\!]^G$.

Note that in both problems we deal with *combined complexity*, as the query is a part of the input.

For measuring complexity, we let $|G|$ denote the size of the graph, $|V|$ the number of nodes in G, and $|\alpha|$ (resp., $|\varphi|$) denote the size of the path expression α (resp., node expression φ). Note that when considering fragments with counting the size of the counter if defined as the number of bits representing it.

The main result of this section is that the combined complexity remains in polynomial time for all fragments we defined in Section 7.1. Not only that, but the exponents are low, ranging from linear to cubic. Notice that for navigational fragments, the low (and even linear) complexity should not come as a surprise. We noticed that $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ is essentially PDL, for which global model checking is known to have linear-time complexity [Alechina and Immerman, 2000,Cleaveland and Steffen, 1993]. Also, polynomial-time combined complexity results are known for pure navigational $\mathsf{GXPath}_{\mathrm{reg}}$ from the PDL perspective as well [Lange, 2006].

Our main contribution is thus to establish the low combined complexity bounds for fragments that handle two new features we added on top of navigational languages: *data value comparisons* and *counters*. The former does increase expressiveness; the latter, as already remarked, does not, but it can make expressions exponentially more succinct. Thus, some work is needed to keep combined complexity polynomial when counters are added.

We first present a general upper bound that shows that combined complexity of both problems is polynomial for the most expressive language we have: regular graph XPath with counting, constant tests, and equality tests.

Theorem 7.2.1. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\#\mathsf{GXPath}_{reg}(c,eq,\sim)$ *can be solved in polynomial time, specifically, i.e.,* $O(|\alpha| \cdot |V|^3)$.

Proof. Both problems can be solved in the required time by a dynamic programming algorithm that processes the parse tree of α in bottom-up fashion and computes, for every path subexpression β of α, the binary relation $[\![\beta]\!]^G$. Similarly, we compute, for every node subexpression φ

of α, the set $[\![\varphi]\!]^G$. Clearly, if each such relation can be computed within time $O(|V|^3)$ (using previously computed relations), both problems can be solved within the required time. We make one exception: we allow $O(|V|^3 \log m)$ time for computing $[\![\beta^{n,m}]\!]^G$ from $[\![\beta]\!]^G$. This is not problematic, since the size of $\beta^{n,m}$ is $O(|\beta| + |\log m|)$.

We discuss how to obtain the desired time bound. The algorithm is similar to an algorithm used for evaluation regular expressions with counters on graphs (Theorem 3.4 in [Losemann and Martens, 2012]).

The base cases for path expressions, that is, computing $[\![\beta]\!]^G$ where β is one of ε, $\_$, a, or a^-, are trivial. Similarly, the base cases for node expressions, that is, computing $[\![\varphi]\!]^G$ where φ is either $\top$, $=c$, or $\neq c$ are trivial as well.

For the induction step we need to consider path expressions of the form $[\varphi]$, $\beta_1 \cdot \beta_2$, $\beta_1 \cup \beta_2$, $\overline{\beta}$, β^*, $\beta^{n,m}$, $\beta_=$, and $\beta_{\neq}$. Also, we need to consider node expressions of the form $\neg\varphi$, $\varphi \wedge \psi$, $\langle\beta\rangle$, $\langle\beta_1 = \beta_2\rangle$, and $\langle\beta_1 \neq \beta_2\rangle$.

In the case of path expressions, the cases $[\varphi]$, $\beta_1 \cup \beta_2$, $\beta_=$, and $\beta_{\neq}$ are trivial because $[\![\varphi]\!]^G$ contains at most $|V|$ elements and $[\![\beta]\!]^G$ at most $|V|^2$ pairs. For example, for $\beta_=$ we can iterate through $[\![\beta]\!]^G$, testing each of its pairs (u,v) and putting it in $[\![\beta_=]\!]^G$ if and only if $\rho(u) = \rho(v)$.

Computing $[\![\beta^*]\!]^G$ amounts to computing the reflexive-transitive closure of $[\![\beta]\!]^G$ which can be done in time $|V|^3$ by Warhsall's algorithm. Computing $[\![\beta^{n,m}]\!]$ within time $O(|V|^3 \log m)$ can be done by fast squaring, as was done in Theorem 3.4 in [Losemann and Martens, 2012].[1] The case $[\![\overline{\beta}]\!]^G$ can be solved by first sorting the pairs from $[\![\beta]\!]^G$ and then performing a single pass over the sorted relation, which costs $O(|V|^2 \log |V|)$ time.

In the case of node expressions the most interesting cases are $\langle\beta_1 = \beta_2\rangle$ and $\langle\beta_1 \neq \beta_2\rangle$. However, computing $[\![\langle\beta_1 = \beta_2\rangle]\!]^G$ and $[\![\langle\beta_1 \neq \beta_2\rangle]\!]^G$ from $[\![\beta_1]\!]^G$ and $[\![\beta_2]\!]^G$ in time $O(|V|^3)$ can be done as follows. For $\langle\beta_1 = \beta_2\rangle$ we need to search if there exist $(v_1, v) \in [\![\beta_1]\!]^G$ and $(v, v_2) \in [\![\beta_2]\!]^G$ such that $\rho(v_1) = \rho(v_2)$. This can be tested in time $O(|V|^3)$ similarly to how one performs a sort-merge join. First, sort relation β_1 and β_2 on the left attribute, which costs time $O(|V|^2 \log |V|)$. Then, for each of the $|V|$ possible values v of the join attribute (in increasing order), we can compute in time $O(|V|)$ the sets $D_{v,1} = \{\rho(v_1) \mid (v, v_1) \in [\![\beta_1]\!]^G\}$ and $D_{v,2} = \{\rho(v_2) \mid (v, v_2) \in [\![\beta_2]\!]^G$. Since both $D_{v,1}$ and $D_{v,2}$ have at most $|V|$ elements, it can be tested in time $O(|V|^2)$ if they have a common data value. The result $[\![\langle\beta_1 = \beta_2\rangle]\!]^G$ contains all v such that $D_{v,1} \cap D_{v,2} \neq \emptyset$ and can therefore be computed in time $O(|V|^3)$. The case $\langle\beta_1 \neq \beta_2\rangle$ is similar. □

The algorithm for Theorem 7.2.1 uses cubic time in $|V|$ because it computes the relations $[\![\beta]\!]^G$ for larger and larger subexpressions β of the given input expression. Therefore, the algorithm uses steps that are at least as difficult as multiplication of $|V| \times |V|$ matrices or computing

[1] Computing $[\![\beta^2]\!]^G$, given $[\![\beta]\!]^G$, takes time $O(|V|^3)$ and, with fast squaring, it costs $O(\log n)$ such operations to compute $[\![\beta^n]\!]^G$. Extending this to $[\![\beta^{n,m}]\!]^G$ is straightforward.

the transitive-reflexive closure of a graph with $|V|$ nodes.

However, if one can avoid computing the relations $[\![\beta]\!]^G$ for subexpressions β, the time bound can be improved.

For the remainder of the section, we assume that there is an ordering on labels of edges and that graphs are represented as adjacency lists such that we can obtain, for a given node v, the outgoing edges or the incoming edges, sorted in increasing order of labels, in constant time. (We note that the linear-time algorithm from [Alechina and Immerman, 2000] for PDL model checking also assumes that adjacency lists are sorted.) The following result is immediate from PDL model checking techniques:

Fact 7.2.2. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}_{reg}^{path\text{-}pos}$ *can be solved in linear time, i.e.,* $O(|\alpha| \cdot |G|)$.

Proof. Since global model checking for PDL is in linear time [Alechina and Immerman, 2000, Cleaveland and Steffen, 1993], it is immediate that QUERY EVALUATION is in time $O(|\alpha| \cdot |G|)$. From this, the same bound for QUERY COMPUTATION can also be derived. Given a query α and a set S, we can mark the nodes in S with a special predicate that occurs nowhere in α. We can then modify query α and use the algorithm for global model checking for PDL to obtain the required output of QUERY COMPUTATION. □

It is straightforward to extend the algorithm of Fact 7.2.2 to c tests, since these can be treated similarly as edge labels.

Corollary 7.2.3. *Both* QUERY EVALUATION *and* QUERY COMPUTATION *problems for* $\mathsf{GXPath}_{reg}^{path\text{-}pos}(\mathsf{c})$ *can be solved in linear time, i.e.,* $O(|\alpha| \cdot |G|)$.

7.3 Expressive power

When gauging the expressive power of query languages the most common yardstick is that of FO [Abiteboul et al., 1995]. Indeed, first-order logic is well established as the core of all relational database queries and it is often one of the query language design goals to achieve some sort of completeness with respect to a fragment of FO. For example one of the governing principles when refining the syntax of the XML query language XPath [ten Cate and Marx, 2007, Kay, 2004] was to make it equivalent to FO over trees, as this provides a well established base for adding new features, while keeping the language compact and easy to understand.

To this end, we will study the expressive power of GXPath and its many dialects when compared to first-order logic. We begin by showing that the core fragment $\mathsf{GXPath}_{\text{core}}$ with no data value comparisons captures FO^3, similarly like its analogue (core XPath 2.0) does on trees. The main difference here is that over trees FO^3 equals full FO, while over graphs this is not the case. After that we also show that for the regular fragment an equivalent statement holds

for FO^3 enriched with binary transitive closure. Following that we move onto data fragments and show that although standard XPath-like data tests fall short of the full power of FO with data value comparisons, the equivalence can be obtained by allowing tests of the kind used in RQDs.

It is important to note that here we compare GXPath only to FO in order to pinpoint the fragments which can be used as a logical kernel of a graph querying language. We will compare GXPath with other graph languages in Chapter 9.

7.3.1 Expressiveness of navigational languages

Here we provide a detailed analysis of expressiveness for navigational features of dialects of GXPath. To understand the expressive power of navigational GXPath we will do two types of comparisons:

- We compare them with FO, fragments and extensions. The core language will capture FO^3. This is similar to a capture result for trees [Marx, 2005]; the main difference is that on graphs, unlike on trees, this falls short of full FO. We also provide a counterpart of this result for $\mathsf{GXPath}_{\mathrm{reg}}$, adding the transitive closure operator.
- We look at the analog of conditional XPath [Marx, 2005] which captures FO over trees and show that, in contrast, over graph databases, it can express queries that are not FO-definable.

Comparisons with FO and relatives To compare expressiveness of GXPath fragments with first-order logic, we need to explain how to represent graph databases as FO structures. Since all the formalisms can express reachability queries (at least with respect to a single label), we view graphs as FO structures

$$G = \langle V, (E_a, E_{a^*})_{a\in\Sigma}\rangle$$

where $E_a = \{(v,v') \mid (v,a,v') \in E\}$ and E_{a^*} is its reflexive-transitive closure.

Recall that FO^k stands for the k-variable fragment of FO, i.e., the set of all FO formulae that use variables from a fixed set $x_1,\ldots,x_k$. As we mentioned, on trees, the core fragment of XPath 2.0 was shown to capture FO^3. We now prove that the same remains true without restriction to trees.

Theorem 7.3.1. *$\mathsf{GXPath}_{core} = \mathrm{FO}^3$ with respect to both path queries and node tests.*

Proof. To prove this we use a result of Tarski and Givant from [Tarski and Givant, 1987] stating that relation algebra with the basis A of binary relations has the same expressive power as first order logic with three variables over the signature A of binary relations and equality.

As we will be using a slight modification of the result found in [Tarski and Givant, 1987] we give precise formulation here. The proof of this version of the result can be found in [Andréka et al., 2001] (see Theorem 1.9 and Theorem 1.10).

First we formalize relation algebras. Let $A = \{R_1, \ldots, R_n\}$ be a set of binary relation symbols. The syntax of relation algebra over A is defined as all expressions built from base relations in A using the operators $\cup, \overline{(\cdot)}, \circ, (\cdot)^-$, denoting union, complement, composition of relations and reverse relation. We are also allowed to use an atomic symbol Id denoting identity.

Our algebra is then interpreted over a structure $M = (V, R_1^M, \ldots, R_n^M)$ where all R_i^M are binary relations over V^2. Interpretations of symbols $\cup, \overline{(\cdot)}, \circ, (\cdot)^-$ and Id is the standard union, complement (with respect to V^2), composition and reverse of binary relations. Id is simply the set of all (v,v) where $v \in V$. We will write $(a,b) \in R^M$, or aR^Mb, when the pair (a,b) belongs to relation R defined over V with relations R_i interpreted as R_i^M.

Theorem 7.3.2 ([Andréka et al., 2001]). *Let $A = \{R_1, \ldots, R_n\}$ be a set of binary relation symbols.*

- *For every expression R in relation algebra $(A, \cup, \overline{(\cdot)}, \circ, (\cdot)^-, Id)$ there is an FO^3 formula in two free variables $\varphi_R(x,y)$, such that for every structure $M = (V, R_1^M, \ldots, R_n^M)$ we have*

$$\{(a,b) : aR^Mb\} = \{(a,b) : M \models \varphi_R[x/a, y/b]\}.$$

- *Conversely, for every FO^3 formula $\varphi(x,y)$, in two free variables, there exists a relation algebra expression R_φ such that for any structure $M = (V, R_1^M, \ldots, R_n^M)$ we have*

$$\{(a,b) : M \models \varphi[x/a, y/b]\} = \{(a,b) : aR_\varphi^Mb\}.$$

Note that we view a graph database $G = (V,E)$ as a structure over the alphabet of binary relations E_a, E_{a^*}, where $a \in \Sigma$. Then a graph database is interpreted as a model

$$M = (V, (E_a^M, E_{a^*}^M) : a \in \Sigma), \text{ where}$$

$$E_a = \{(v,v') : (v,a,v') \in E\}$$

and E_{a^*} is its reflexive transitive closure. Note that the Tarski-Givant result states something stronger, namely that the equivalence will hold over any structure, no matter if a^* is interpreted as the transitive closure of a or not. This means that it will in particular hold on all the structures where it is, and those are our graph databases.

First we give a translation from $\mathsf{GXPath}_{\text{core}}$ into FO^3. That is, for every path expression e, we provide a formula $F_e(x,y)$ in two free variables such that for, any graph database $G = (V,E)$, we have $[\![e]\!]^G = \{(v,v') \in G : M \models F_e[x/v, y/v']\}$, where $M = (V, (E_a^M, E_{a^*}^M) : a \in \Sigma)$ and $E_a = \{(v,v') : (v,a,v') \in E\}$ and E_{a^*} its reflexive transitive closure. Similarly, for every node expression φ, we define a formula $F_\varphi(x)$ in one free variable. The definition is by simultaneous induction on the structure of $\mathsf{GXPath}_{\text{core}}$ expressions.

Base cases:

- $e = a$ then $F_e(x,y) \equiv E_a(x,y)$
- $e = a^*$ then $F_e(x,y) \equiv E_{a^*}(x,y)$
- $e = a^-$ then $F_e(x,y) \equiv E_a(y,x)$
- $e = (a^-)^*$ then $F_e(x,y) \equiv E_{a^*}(y,x)$
- $\varphi = \top$ then $F_e(x) \equiv x = x$.

Inductive cases:

- $e = [\varphi]$ the $F_e(x,y) \equiv (x = y) \wedge F_\varphi(x)$
- $e = \alpha \cdot \beta$ then $F_e(x,y) \equiv \exists z(F_\alpha(x,z) \wedge \exists x(x = z \wedge F_\beta(x,y)))$
- $e = \alpha \cup \beta$ then $F_e(x,y) \equiv F_\alpha(x,y) \vee F_\beta(x,y)$
- $\varphi = \neg\psi$ then $F_\varphi(x) \equiv \neg F_\psi(x)$
- $\varphi = \psi \wedge \psi'$ then $F_\varphi(x) \equiv F_\psi(x) \wedge F_{\psi'}(x)$
- $\varphi = \langle\alpha\rangle$ then $F_\varphi(x) \equiv \exists y F_\alpha(x,y)$
- $e = \overline{\alpha}$ then $F_e(x,y) \equiv \neg F_\alpha(x,y)$.

The claim easily follows. Note that we have shown that our expressions can be converted into FO^3 over a fixed interpretation of relation symbols appearing in our alphabet (that is when $E_{a^*} = (E_a)^*$). The result by Tarski and Givant is stronger, since it holds for any interpretation. Note that this does not invalidate our result, since we are interested only in this fixed interpretation of graph predicates.

To prove the equivalence of $\mathsf{GXPath}_{\text{core}}$ with FO^3 we now show that every relation algebra expression has an equivalent $\mathsf{GXPath}_{\text{core}}$ path expression.

First we show how to convert every relation algebra query into an equivalent $\mathsf{GXPath}_{\text{core}}$ expression over graph databases. To be more precise, we show that for any relation algebra expression R over the signature $(E_a, E_{a^*})_{a\in\Sigma}$ there is a path expression e_R of $\mathsf{GXPath}_{\text{core}}$such that for any graph database $G = (V,E)$ it holds that $[\![e_R]\!]^G = \{(a,b) \in R^M\}$. Here M is obtained from G as before. In particular we assume that E_{a^*} is the reflexive transitive closure of E_a. We do this inductively on the structure of RA expressions R.

Base cases:

- If $R = E_a$ then $e_R = a$.

- If $R = E_{a^*}$ then $e_R = a^*$.
- If $R = Id$ then $e_R = \varepsilon$.

Inductive cases:

- If $R = R_1 \cup R_2$ then $e_R = e_{R_1} \cup e_{R_2}$.
- If $R = R_1 \circ R_2$ then $e_R = e_{R_1} \cdot e_{R_2}$.
- If $R = S^-$ then $e_R = (e_S)^-$.
- If $R = \overline{S}$ then $e_R = \overline{e_S}$.

To show the equivalence between $R = S^-$ and $e_R = (e_S)^-$ we need the following claim.

Claim 7.3.3. *For every GXPath$_{core}$ path expression e there is a GXPath$_{core}$ expression e^- such that $[\![e^-]\!]^G = \{(v,v') : (v',v) \in [\![e]\!]^G\}$, for every graph G.*

The proof of this is just an easy induction on expressions. We simply push the reverse onto atomic statements. Note that this is the reason why we can not simply drop the converse operators from our syntax.

All the other equivalences follow from the definition and the inductive hypothesis.

Now let $\varphi(x,y)$ be an arbitrary FO^3 formula. By Theorem 7.3.2 we know that there is a relation algebra expression R_φ equivalent to φ over all structures that interpret $\{E_a, E_{a^*} : a \in \Sigma\}$. In particular it is true over all the structures where $E_{a^*} = (E_a)^*$. By the previous paragraph we know that there is a GXPath$_{\text{core}}$ expression e_{R_φ} equivalent to R_φ.

In particular this means that for every graph database $G = (V,E)$ it holds that for the model $M = (V, (E_a, E_{a^*}) : a \in \Sigma)$, derived from G, we have the following:

$$\{(a,b) : M \models \varphi[x/a, y/b]\} = \{(a,b) : (a,b) \in R_\varphi^M\}.$$

On the other hand, we also have:

$$[\![e_{R_\varphi}]\!]^G = \{(a,b) : (a,b) \in R_\varphi^M\}.$$

Thus we conclude that

$$\{(a,b) : M \models \varphi[x/a, y/b]\} = [\![e_{R_\varphi}]\!]^M.$$

The previous part shows equivalence between path expressions and formulas with two free variables. To deal with formulas with a single free variable $F(x)$ we do the following. Define $F'(x,y) = x = y \wedge F(x)$. Note that F' selects all pairs (v,v) such that $F(v)$ holds. Now find an equivalent path expression α (we know we can do this by going through relation algebra) and let $e = \langle \alpha \rangle$. □

Not all results about the expressiveness of XPath on trees extend to graphs. For instance, on trees, the regular fragment with no negation on paths (i.e., the path-positive fragment) can express all of FO [Marx, 2005]. This fails over graphs: $\mathsf{GXPath}_{\mathrm{reg}}$ fails to express even all of FO^2 when restricted to its path-positive fragment (i.e, the fragment that still permits unary negation).

Proposition 7.3.4. *There exists a binary* FO^2 *query that is not definable in* $\mathsf{GXPath}_{reg}^{path\text{-}pos}$.

Proof. The idea is to observe that path-positive fragments of GXPath cannot define the universal binary relation on an input graph. The query not definable in $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ is then the one saying that there are at least two nodes in a given graph.

Formally, let $\psi(x,y) \equiv \exists x \exists y(\neg x = y)$. It is easy to see that $[\![\psi]\!]^G = \{(x,y) : (x,y) \in V^2\}$ if $G = \langle V, E\rangle$ has at least two nodes and $[\![\psi]\!]^G = \emptyset$ otherwise. (Notice that the variables x,y in ψ are immediately "overwritten" by the existential quantification.)

Consider the graphs $G_1 = \langle \{v,v'\}, \emptyset\rangle$ and $G_2 = \langle \{v\}, \emptyset\rangle$. That is, we have no edges. It follows that $[\![\psi]\!]^{G_1} = \{(v,v'),(v',v)\}$ and $[\![\psi]\!]^{G_2} = \emptyset$. It can be shown by induction on the structure of path $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ expressions that we either have that $[\![\alpha]\!]^{G_1} = \{(v,v),(v',v')\}$ and $[\![\alpha]\!]^{G_2} = \{(v,v)\}$, or $[\![\alpha]\!]^{G_1} = \emptyset$ and $[\![\alpha]\!]^{G_2} = \emptyset$. Similarly for node expressions it can be shown that either $[\![\varphi]\!]^{G_1} = G_1$ and $[\![\varphi]\!]^{G_2} = G_2$, or $[\![\varphi]\!]^{G_1} = \emptyset$ and $[\![\varphi]\!]^{G_2} = \emptyset$. □

We now move to $\mathsf{GXPath}_{\mathrm{reg}}$ and relate it to a fragment of FO^*, the parameter-free fragment of the transitive-closure logic. The language of FO^* extends the one of FO with a transitive closure operator that can be applied to formulas with precisely two free variables. That is, for any FO formula $F(x,y)$, the formula $F^*(x,y)$ is also an FO^* formula. The semantics is the reflexive-transitive closure of the semantics of F. That is, $G \models F^*(a,b)$ iff $a = b$ or there is a sequence of nodes $a = v_0, v_1, \ldots, v_n = b$ for $n > 0$ such that $G \models F(v_i, v_{i+1})$ whenever $0 \leq i < n$.

By $(\mathrm{FO}^*)^k$ we mean the k-variable fragment of FO^*. Note that when we deal with FO^* and $(\mathrm{FO}^*)^k$, we can view graphs as structures of the vocabulary $(E_a)_{a\in\Sigma}$, since all the E_{a^*}s are definable, and there is no reason to include them in the language explicitly.

Over trees, regular XPath is known to be equal to $(\mathrm{FO}^*)^3$ [ten Cate, 2006]. The next theorem shows that over graphs, these logics coincide as well.

Theorem 7.3.5. $\mathsf{GXPath}_{reg} = (\mathrm{FO}^*)^3$.

Proof. The containment of $\mathsf{GXPath}_{\mathrm{reg}}$ in $(\mathrm{FO}^*)^3$ is done by a routine translation.

To show the converse, we use techniques similar to those in the proof of Theorem 7.3.1: we extend $(\mathrm{FO}^*)^3$ and relation algebra equivalence to state that relation algebra with the transitive closure operator has equal expressive power to $(\mathrm{FO}^*)^3$ over the class of all labeled graphs. For this one can simply check that the inductive proof from [Andréka et al., 2001] can be extended by adding two extra inductive clauses. Namely, when going from relation algebra to FO^3 we

simply state that expressions of the form R^* are equivalent to $F_R^*(x,y)$, where F_R is the formula equivalent to R. In the other direction we simply state that $F^*(x,y)$ is equivalent to $(R_F(x,y))^*$. Here by $R_F(x,y)$ we denote the expression equivalent to $F(x,y)$, when the variables are used in that particular order. After that one verifies that the correctness proof of [Andréka et al., 2001] applies. □

What about the relative expressive power of $\mathsf{GXPath}_{\text{core}}$ and $\mathsf{GXPath}_{\text{reg}}$? For positive fragments, known results on trees (see [ten Cate and Marx, 2007]) imply the following.

Corollary 7.3.6. $\mathit{GXPath}_{core}^{pos} \subsetneq \mathit{GXPath}_{reg}^{pos}$.

We shall now see that the strict separation applies to full languages. This is not completely straightforward even though $\mathsf{GXPath}_{\text{core}}$ is equivalent to a fragment of FO, since the latter uses the vocabulary with transitive closures. This makes it harder to apply standard techniques, such as locality, directly. We shall see how to establish separation by taking a detour through conditional XPath.

Conditional GXPath It was shown in [Marx, 2005] that to capture FO over XML trees, one can use *conditional* XPath, which essentially adds the temporal *until operator*. That is, it expands the core-XPath's a^* with $(a[\varphi])^*$, which checks that the test $[\varphi]$ is true on an a-labeled path. Formally, its path formulae are given by:

$$\alpha,\beta \quad := \quad \varepsilon \mid \_ \mid a \mid a^- \mid a^* \mid a^{-*} \mid (a[\varphi])^* \mid (a^-[\varphi])^* \mid [\varphi] \mid \alpha\cdot\beta \mid \alpha\cup\beta \mid \overline{\alpha}$$

We refer to this language as $\mathsf{GXPath}_{\text{cond}}$. We now show that the FO capture result fails rather dramatically over graphs: there are even positive $\mathsf{GXPath}_{\text{cond}}$ queries not expressible in FO.

Theorem 7.3.7. *There is a* $\mathit{GXPath}_{cond}^{pos}$ *query not expressible in* FO.

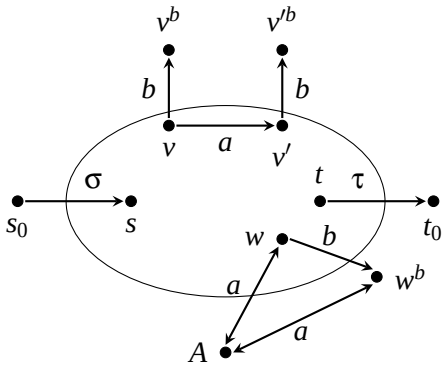
Note that the standard inexpressibility tools for FO, such as locality, cannot be applied straightforwardly since the vocabulary of graphs already contains all the transitive closures E_{a^*}; in fact this means that in $\mathsf{GXPath}_{\text{cond}}^{\text{pos}}$ the query asking for transitive closures of base relations is trivially definable, even though it is not definable in FO over the E_as. So the way around this is to combine locality with the composition method: we use locality to establish a winning strategy for the duplicator in a game that does not involve transitive closures, and then use composition to extend the winning strategy to handle transitive closures.

Proof. To prove this we will need several auxiliary results.

Let $\Sigma = \{a,b,\sigma,\tau\}$ be an alphabet of labels. We define a class $\mathcal{C}$ of Σ-labeled graphs as follows.

Take any graph $G = (V,E)$ over the singleton alphabet $\{a\}$ of labels. Fix two nodes s and t in G. Let $G^{\mathcal{C}}(s,t)$ be the graph obtained from G as follows. First, it contains all the nodes and

edges of G. For every node $v \neq s,t$ in G we add a new node v^b and an edge (v,b,v^b) to $G^{\mathcal{C}}(s,t)$. We also add two new nodes, s_0 and t_0, together with edges (s_0,σ,s) and (t,τ,t_0), coming into s and leaving t. These nodes and edges are added to distinguish s and t in our graph. Finally, we add one extra node called A, and for every other node in $G^{\mathcal{C}}(s,t)$ we add two edges, one going into A and the other returning from A to the same node, both labeled a. Now add this $G^{\mathcal{C}}(s,t)$ to $\mathcal{C}$. The modifications are illustrated in the following image.



Also define $\mathcal{C}^-$ to be the class of graphs that are obtained from the graphs in $\mathcal{C}$ by removing the node A and all the associated edges.

Now let the property P stand for

- t is reachable from s via a path labeled with $(a[b])^*$.

That is, t is reachable from s by a path that proceeds forwards by a-labeled edges, but also has to have a b labeled edge leaving every internal node on the path.

To obtain the desired result we will first prove the following claim.

Claim 7.3.8. *The property P is not expressible in* FO *in vocabulary $\{E_a,E_b,E_\sigma,E_\tau,E_{a^*},E_{b^*},E_{\sigma^*},E_{\tau^*}\}$ over the class $\mathcal{C}$. Here, as before, we assume that E_{ℓ^*} is the reflexive transitive closure of E_ℓ, for $\ell \in \{a,b,\sigma,\tau\}$.*

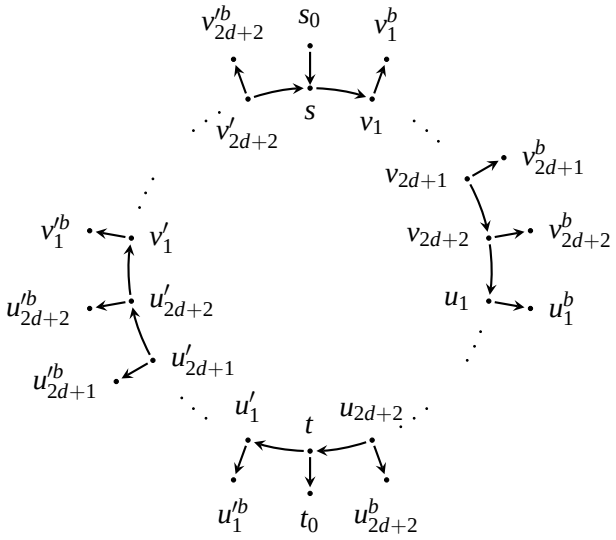
To prove this claim we will use Hanf-locality and composition of games. For the proof we use three lemmas.

In the first one, we use the standard notion of a *neighborhood* of an element in a structure, and the notion of Hanf-locality. For details, see [Libkin, 2004]. Specifically, for two graphs G^1,G^2, we write $G^1 \leftrightarrows_d G^2$ if there is a bijection f between nodes of G^1 and nodes of G^2 (in particular, the sets of nodes must have the same cardinality) such that the radius-d neighborhoods of each node v in G^1 and $f(v)$ in G^2 are isomorphic. The radius-d neighborhood around v is the substructure generated by all nodes reachable from v by a path (using all types of edges) of length at most d.

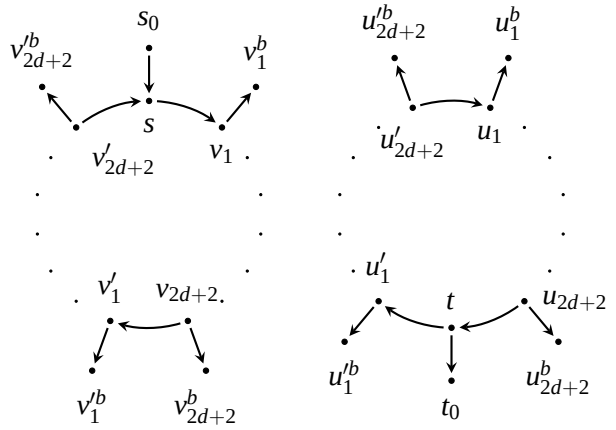
Locality is meaningless over structures in $\mathcal{C}$, since every two nodes are connected by a path of length 2, so $\leftrightarrows_2$ is isomorphism. This is why we get the result in several steps.

Lemma 7.3.9. *For every $d \geq 0$ there exist two graphs G_d^1 and G_d^2, as structures of the vocabulary $\{E_a, E_b, E_\sigma, E_\tau\}$, in $\mathcal{C}^-$ such that $G_d^1 \leftrightarrows_d G_d^2$ and G_d^1 satisfies P, while G_d^2 does not.*

Proof. To see this take arbitrary d and let the graphs G_d^1 and G_d^2 be as in the following two images. All the labels on the circles are a, the incoming edges from s_0s to the ss are labeled σ, the outgoing edges from the ts to t_0s are labeled τ, and the edges from the vs to the v^bs are labeled b.



Graph G_d^1



Graph G_d^2

Now let $f : G_d^1 \to G_d^2$ be the bijection defined by the node labels in the natural way: each node gets mapped to the one with the same name in the other graph. That is we set $f(s) := s$, $f(v_i) := v_i$, then $f(v_i^b) := v_i^b$ and similarly for v'_i, u_i, etc.

To see that $G_d^1 \leftrightarrows_d G_d^2$ we have to check $N_d^{G_d^1}(c) \cong N_d^{G_d^2}(f(c))$ for every c. But this is now easily established, since the d neighborhood of any c and $f(c)$ will simply be extended chains of length d around c and $f(c)$. In particular, it is possible that they intersect the d neighborhood

of either s or t, but never both. We thus conclude that they will always be isomorphic, giving us the desired result. □

Now from Lemma 7.3.9 and Corollary 4.21 in [Libkin, 2004], which shows that Hanf-locality with a sufficiently large radius implies the winning strategy for the duplicator in an Ehrenfeucht-Fraïssé game, we obtain the following.

Lemma 7.3.10. *For every $m \geq 0$ there exists $d \geq 0$ so that $G^1_d \equiv_m G^2_d$.*

As usual, by $\equiv_m$ we denote the fact that duplicator has a winning strategy in an m-round Ehrenfeucht-Fraïssé game. This game is still played on structures in the vocabulary that does not use transitive closures.

Now let $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ be obtained from G^1_d and G^2_d by adding, as in the picture above, a node A with a-edges to and from every other node. We view these graphs as structures of the vocabulary that has all the relations E_ℓ and E_{ℓ^*} for each of the four labels ℓ we have. Next, we show

Lemma 7.3.11. *If $G^1_d \equiv_m G^2_d$, then $\mathbf{G}^1_d \equiv_m \mathbf{G}^2_d$.*

The strategy is very simple: the duplicator plays by copying the moves from the game $G^1_d \equiv_m G^2_d$ as long as the spoiler does not play the A-node. If the spoiler plays the A-node in one structure, the duplicator responds with the A-node in the other. We now need to show that this preserves all the relations. Clearly this strategy preserves all the relations E_ℓ among nodes other than the A-node, simply by assumption. Moreover, since $E_{\ell^*} = E_\ell$ for $\ell \neq a$, we have preservation of the transitive closures other than that of E_a as well. So we need to prove that the strategy preserves E_{a^*}, but this is immediate since in both graphs E_{a^*} is interpreted as the total relation. This proves the lemma.

The claim now follows from the lemmas: assume that P is expressible in FO, over the full vocabulary, by a formula of quantifier rank m. Pick sufficiently large d to ensure that $\mathbf{G}^1_d \equiv_m \mathbf{G}^2_d$. Then $\mathbf{G}^1_d$ and $\mathbf{G}^2_d$ must agree on P, but they clearly do not, since the extra paths introduced in these graphs compared to G^1_d and G^2_d go via the A-node, which does not have a b-successor.

Now to prove Theorem 7.3.7 consider a conditional graph XPath expression $\sigma(a[b])^*[\tau]$. Over graphs as considered here it defines precisely the property P, which, as just shown, is not FO-expressible in the full vocabulary. □

We can now fulfill our promise and establish separation between $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$. Since $\mathsf{GXPath}_{\mathrm{core}} \subseteq$ FO and we just saw a conditional (and thus regular) GXPath query not expressible in FO, we have:

Corollary 7.3.12. *$\mathsf{GXPath}_{core} \subsetneq \mathsf{GXPath}_{reg}$.*

7.3.2 Expressiveness of data languages

We saw that for navigational features, core graph XPath captures FO^3. The question is whether this continues to be so in the presence of data tests. First, we need to explain how to describe data graphs as FO-structures to talk about FO with data tests.

Following the standard approach for data words and data trees [Segoufin, 2007], we do so by adding a binary predicate for testing if two nodes hold the same data value. That is, a data graph is then viewed as a structure $G = \langle V, (E_a, E_{a^*})_{a\in\Sigma}, \sim\rangle$ where $v \sim v'$ iff $\rho(v) = \rho(v')$. To be clear that we deal with FO over that vocabulary, we shall write $\mathrm{FO}(\sim)$. If we want to talk about constant data tests (i.e., $=c$), we make the language two-sorted, adding another domain for data values and using a separate set of constant symbols. In that case we shall refer to $\mathrm{FO}(\mathsf{c}, \sim)$.

It turns out that the equivalence with FO^3 breaks when we consider XPath style data tests.

Theorem 7.3.13.
- $\mathit{GXPath}_{core}(eq) \subsetneq \mathrm{FO}^3(\sim)$;
- $\mathit{GXPath}_{core}(c, eq) \subsetneq \mathrm{FO}^3(c, \sim)$.

Proof sketch. The first containment uses the translation into FO^3 shown in the proof of Theorem 7.3.1. For the new data operators, we use the following. If $e = \langle \alpha = \beta \rangle$ then

$$F_e(x) \equiv \exists y, z(y \sim z \wedge F_\alpha(x,y) \wedge \exists y(z = y \wedge F_\beta(x,y)))$$

and likewise for the inequality comparison.

Translation of constants is self-evident.

To prove strictness we show that the FO^3 query $F(x,y) \equiv x \sim y$ is not definable in $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq})$. Note that F defines the set of all pairs of nodes carrying the same data value. The proof of this is implicit in the proof of Proposition 7.3.14. □

Thus, the standard XPath data tests are insufficient for capturing FO^3 over data graphs. This naturally leads to a question: what can be added to data tests to capture the full power of FO^3? The answer, as it turns out, is quite simple: we need to use the same sort of data value tests as in RQDs.

Recall that these are defined by adding two expressions to the grammar for α: one is $\alpha_=$, the other is $\alpha_\neq$. Semantics, over data graphs, is

$$\begin{aligned} [\![\alpha_=]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) = \rho(v')\} \\ [\![\alpha_\neq]\!]^G &= \{(v,v') \in [\![\alpha]\!]^G \mid \rho(v) \neq \rho(v')\} \end{aligned}$$

In other words, we test whether data values at the beginning and at the end of a path are the same, or different. As mentioned before, such an extension is denoted by $\sim$, i.e. we talk about languages $\mathsf{GXPath}(\sim)$ (with the usual sub- and superscripts).

The first observation is that these tests indeed add to the expressiveness of the languages.

Proposition 7.3.14. *The path query $a_=$, for $a \in \Sigma$, is not definable in $\mathit{GXPath}_{reg}(c, eq)$.*

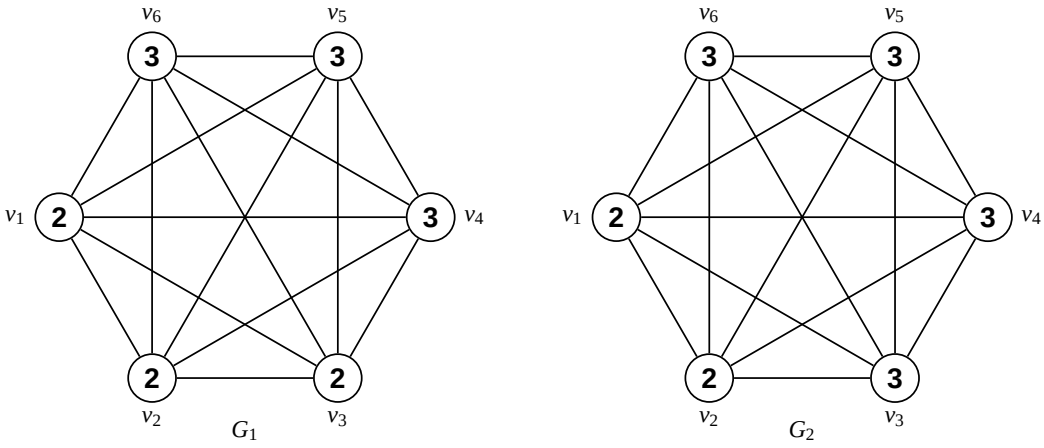
Note that this query, $a_=$, is definable on trees by the $\mathsf{GXPath}_{\text{core}}(\mathsf{eq})$ query $[\langle \varepsilon = a \rangle] \cdot a \cdot [\langle \varepsilon = a^- \rangle]$. This is because the parent of a given node is unique. However, on graphs this is not always the case, and thus new equality tests add power.

Proof. Here we prove that even though $\mathsf{GXPath}_{\text{reg}}(\mathsf{c},\mathsf{eq})$ can test if a node has an a-successor with the same data value by the means of expression $\langle \varepsilon = a \rangle$, which will return the set $\{v \in V \mid \exists v' \in V \text{ and } (v,v') \in [\![a_=]\!]^G\}$, it has no means of retrieving that specific successor.

We will first prove the result without constant tests.

To prove that $a_=$ is not expressible in $\mathsf{GXPath}_{\text{reg}}(\mathsf{eq})$ over graphs we will give two graphs G_1 and G_2, such that $[\![a_=]\!]^{G_1} \neq [\![a_=]\!]^{G_2}$, but for every $\mathsf{GXPath}_{\text{reg}}(\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$.

Both G_1 and G_2 will be the graphs K_6, that is the complete graphs with six vertices and with data values $2,2,2,3,3,3$ and $2,2,3,3,3,3$, respectively, attached to the nodes. All the edges in both G_1 and G_2 are labelled a. The graphs G_1 and G_2 are pictured in the following image.



It follows from the definitions that $(v_2,v_3) \in [\![a_=]\!]^{G_1}$, while $(v_2,v_3) \notin [\![a_=]\!]^{G_2}$. We conclude that $[\![a_=]\!]^{G_1} \neq [\![a_=]\!]^{G_2}$.

We now show that for any $\mathsf{GXPath}_{\text{reg}}(\mathsf{eq})$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. In particular we show the following:

- For any path query α one of the following holds:
  - $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$, or
  - $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$, or
  - $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = {G_1}^2$, or
  - $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = {G_1}^2 - Id(G_1)$.
- For any node query φ one of the following holds:
  - $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$, or
  - $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.

As before $Id(G_1)$ stands for the set $\{(x,x) : x \in G_1\}$. Note that since the sets of nodes of G_1 and G_2 are the same (and the graphs are not isomorphic because of the different data values), we can write $[\![\varphi]\!]^{G_2} = G_1$ and other claims.

We prove this claim by induction on the structure of our $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$ expression e.

The base cases trivially follow. For the induction step assume that our claim is true for the expressions of lower complexity. We proceed by cases.

- If $\alpha = [\varphi]$ then by the inductive hypothesis we have two cases.
  - Either $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$, in which case $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$,
  - Or $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$, in which case $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$.
- If $\alpha = \alpha' \cup \beta'$ then the claim follows from the induction hypothesis and the fact that the set $\{\emptyset, {G_1}^2, {G_1}^2 - Id(G_1), Id(G_1)\}$ is closed under taking unions.
- If $\alpha = \alpha' \cdot \beta'$ we proceed as follows.

  Note first that $[\![\alpha]\!]^{G_1} = \emptyset$ iff $[\![\alpha']\!]^{G_1} = \emptyset$ or $[\![\beta']\!]^{G_1} = \emptyset$ (this follows from the inductive hypothesis about the structure of the answers, since for any other case the sets have nonempty composition). This is now equivalent to the same being true in G_2 and thus to $[\![\alpha]\!]^{G_2} = \emptyset$.

  If $[\![\alpha]\!]^{G_1} \neq \emptyset$ then we know that both $[\![\alpha']\!]^{G_1}$ and $[\![\beta']\!]^{G_1}$ belong to $\{{G_1}^2, {G_1}^2 - Id(G_1), Id(G_1)\}$. The claim now simply follows from the inductive hypothesis and the fact that the set $\{{G_1}^2, {G_1}^2 - Id(G_1), Id(G_1)\}$ is closed under composition of relations.
- If $\alpha = \overline{\alpha'}$ we have four cases.
  - In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = \emptyset$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = {G_1}^2$.
  - In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = {G_1}^2$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$.
  - In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = {G_1}^2 - Id(G_1)$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$.
  - In case that $[\![\alpha']\!]^{G_1} = [\![\alpha']\!]^{G_2} = Id(G_1)$ we have $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = {G_1}^2 - Id(G_1)$.
- If $\alpha = \alpha'^*$ we have the same situation as in the previous case. In particular we know that transitive closures in each case will be the same.
- If $\varphi = \neg\varphi$ we have the following.
  - In case that $[\![\varphi']\!]^{G_1} = [\![\varphi']\!]^{G_2} = G_1$ we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.
  - In case that $[\![\varphi']\!]^{G_1} = [\![\varphi']\!]^{G_2} = \emptyset$ we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.
- If $\varphi = \varphi' \wedge \psi'$ the claim easily follows.
- If $\varphi = \langle\alpha\rangle$ we consider the value of $[\![\alpha]\!]^{G_1}$.

- In case that $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = \emptyset$ we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.
- In case that $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = {G_1}^2, Id(G_1)$, or ${G_1}^2 - Id(G_1)$ we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.

- If $\varphi = \langle \alpha = \beta \rangle$ we proceed by cases, depending of the value of $[\![\alpha]\!]^{G_1}$ and $[\![\beta]\!]^{G_1}$.

  Note that if either equals $\emptyset$ we get that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$. There are now nine possible cases remaining.

  1. $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = Id(G_1)$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.
  2. $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = {G_1}^2$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.
  3. $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = {G_1}^2 - Id(G_1)$ implies that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$.
  4. All the remaining cases have the same result.

- If $\varphi = \langle \alpha \neq \beta \rangle$ we proceed by cases, depending of the value of $[\![\alpha]\!]^{G_1}$ and $[\![\beta]\!]^{G_1}$.

  Note that if either equals $\emptyset$ we get that $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$. Just as for $\langle \alpha = \beta \rangle$ we have nine cases. It is easily verified that we have $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = G_1$ for each case, except when $[\![\alpha]\!]^{G_1} = [\![\alpha]\!]^{G_2} = Id(G_1)$ and $[\![\beta]\!]^{G_1} = [\![\beta]\!]^{G_2} = Id(G_1)$. In this case we get $[\![\varphi]\!]^{G_1} = [\![\varphi]\!]^{G_2} = \emptyset$.

To extend the induction to work for constants, we assume the contrary. Let the e be an expression defining $a_=$. We exchange the data values 2 and 3 in our graphs G_1 and G_2 by any two data values that do not appear as constants in e. The proof is now the same as in the case without constants.

This completes the proof. □

With the extra power given to us by the equality tests, we can capture FO^3 over data graphs.

Theorem 7.3.15. $\mathit{GXPath}_{core}(\sim) = \mathrm{FO}^3(\sim)$.

Proof. We follow the technique of the proof of Theorem 7.3.1. All of the translations used there still apply. The proof that relation algebra is contained in the language $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ is the same as without data values. We only have to add conversion of the new symbol $\sim$: if $R = \sim$, then $e = \varepsilon \cup (\overline{\varepsilon})_=$.

For the other direction we have to show how to translate new path expressions $\alpha_=$ and $\alpha_{\neq}$ into $\mathrm{FO}^3(\sim)$. This is done as follows: if $e = \alpha_=$ then $F_e(x,y) \equiv F_\alpha(x,y) \wedge x \sim y$ and likewise for inequality. The equivalences easily follow. Now the theorem follows from the equivalence of relation algebra and FO^3 [Tarski and Givant, 1987]. □

By adopting the technique used in Theorem 7.3.5 it is straightforward to see that the previous result extends to $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$.

Theorem 7.3.16. $\mathit{GXPath}_{\mathit{reg}}(\sim) = (\mathrm{FO}^*)^3(\sim)$.

As mentioned before, one could also allow constant tests in the language. It is then easy to see that the equivalence extends to FO with constants.

Corollary 7.3.17.

- $\mathit{GXPath}_{\mathit{core}}(c, \sim) = \mathrm{FO}^3(c, \sim)$.
- $\mathit{GXPath}_{\mathit{reg}}(c, \sim) = (\mathrm{FO}^*)^3(c, \sim)$.

7.4 Hierarchy of the fragments

By coupling the basic navigational languages – $\mathsf{GXPath}_{\mathrm{core}}$ and $\mathsf{GXPath}_{\mathrm{reg}}$ – with various possibilities of data tests, such as no data tests, constant tests, XPath-style equality tests, RQD equality tests, or all of them, we obtain sixteen languages, ranging from $\mathsf{GXPath}_{\mathrm{core}}$ to $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq}, \sim)$. Recall that adding counting does not affect expressiveness, only the complexity of query evaluation.

The question is then, how do these fragments compare to each other?

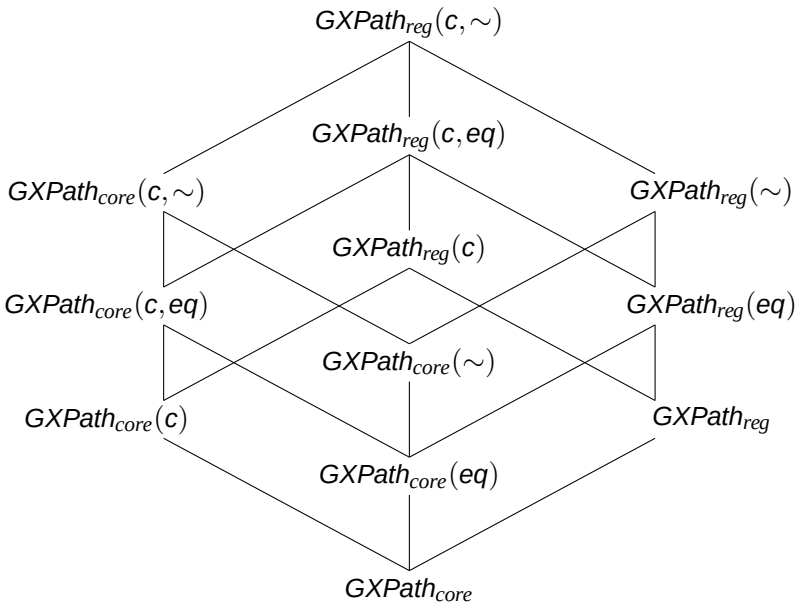
First thing we note is that some of the fragments collapse. Namely, from Theorem 7.3.15 we know that every $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$ query can be expressed in $\mathsf{GXPath}_{\mathrm{core}}(\sim)$, and the same holds for regular fragments using Theorem 7.3.16.

To perform such a transformation explicitly we simply need to show how to convert every test of the form $\langle \alpha = \beta \rangle$ to one using only $=$ comparisons from $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ and that the same can be done for inequality. It is not difficult to see that every node expression of the form $\langle \alpha = \beta \rangle$ is equivalent to $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ expression $\langle \alpha \cdot (\alpha^- \cdot \beta)_= \cdot \beta^- \cap \varepsilon \rangle$, and similarly for $\neq$.

Therefore we can conclude that any fragment where both eq and $\sim$ data tests are present collapses to the one with only $\sim$. For example $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq}, \sim)$ is the same as $\mathsf{GXPath}_{\mathrm{core}}(\sim)$ and so on, bringing the number of possible fragments to twelve.

Next we establish the full hierarchy of the remaining fragments.

Theorem 7.4.1. *The relative expressive power of graph XPath languages with data comparisons is as shown below:*



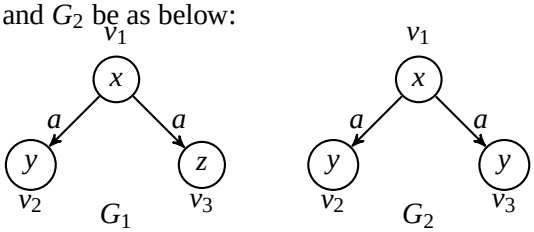
Here a line upwards means that the lower fragment is strictly contained in the upper other, while the lack of the line means that the fragments are incomparable.

Proof. The result follows from Corollary 7.3.12 (for navigational fragments), the fact that $\sim$ comparisons subsume usual XPath-style tests, and the following two observations which show that c tests and eq or $\sim$ tests are not mutually definable. Namely, take an alphabet Σ containing letter a. Let c be a fixed data value. Then:

- There is no $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ expression equivalent to the $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{c})$ query $q_{\mathsf{c}} := (= c)$.
- There is no $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expression equivalent to the $\mathsf{GXPath}_{\mathrm{core}}(\mathsf{eq})$ query $q_{\mathsf{eq}} := \langle a \neq a \rangle$.

For the first item, simply take two single-node data graphs G_1 and G_2, with G_1's single node holding value c, and G_2 holding a different value c'. Hence, $[\![q_{\mathsf{c}}]\!]^{G_1}$ selects the only node of G_1, while $[\![q_{\mathsf{c}}]\!]^{G_2} = \emptyset$. However, a straightforward induction on the structure of expressions shows that for every $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$ query e we have $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$.

For the second item assume that there is an $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expression ex equivalent to q_{eq}. Take any three pairwise distinct data values x, y, z that are different from all the constants appearing in ex and let G_1 and G_2 be as below:

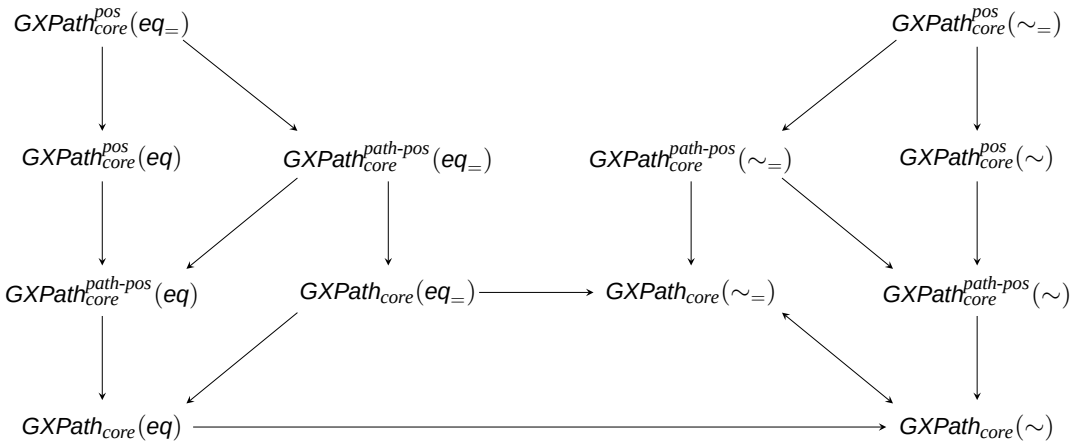


One can show by straightforward induction on $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expressions e that use only constants appearing in ex that $[\![e]\!]^{G_1} = [\![e]\!]^{G_2}$. Thus, q_{eq} cannot be a $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c})$ expression, since $[\![q_{\mathsf{eq}}]\!]^{G_1} \neq [\![q_{\mathsf{eq}}]\!]^{G_2}$.

Note that this also shows that $\mathsf{GXPath}_{\text{core}} \subsetneq \mathsf{GXPath}_{\text{core}}(\mathsf{c})$ and $\mathsf{GXPath}_{\text{core}} \subsetneq \mathsf{GXPath}_{\text{core}}(\mathsf{eq})$. □

As shown in Proposition 7.1.3, the path positive and the positive fragments are strictly contained in the full language. When comparing various graph languages later in Chapter 9 we will also show that the positive fragment can not express node negation (see Theorem 9.2.3). Furthermore, when considering query containment problem in Chapter 10 it will be important to distinguish between fragments that use explicit inequality comparisons from the ones that compare data values for equality only. A subfragment of a $\sim$ fragment using only equalities (that is subexpressions of the form $\alpha_{\neq}$ are not permitted) will be denoted by $\sim_{=}$, while the corresponding subfragment of a eq fragment will be denoted by $\mathsf{eq}_{=}$. The following theorem establishes the hierarchy of such fragments. It is important to note here that in the absence of path negation one can no longer simulate eq tests using the $\sim$ tests. Note that in order to avoid notational clutter we disregards constants in this comparison.

Theorem 7.4.2. *The relative expressive power of GXPath_{core} fragments based on restricting negation in navigational features or data comparisons is given below.*



Here a line from one fragment to another signifies that the source fragment is contained in the target one. An analogous set of results holds for GXPath_{reg}.

Proof. As just discussed, the positive fragments are strictly contained in the path-positive ones. Furthermore, by Proposition 7.1.3 we know that the path-positive fragments are strictly contained in the full language allowing negation over paths.

From Theorems 7.3.15 and 7.3.13 we also get that when path negation is present $\sim$ fragments subsume the ones with eq tests.

To show that $\mathsf{eq}_{=}$ fragment is contained in the eq we simply need to take a graph G_1 with two nodes holding the same data value, connected by an a-labelled edge in both directions and a graph G_2, this time with two nodes holding different data values, again connected by

a-labelled edges. Both graphs also have self loops labelled a for each node. A straightforward induction on $\mathsf{GXPath}_{\text{core}}(\mathsf{eq}_=)$ expressions shows that the result of any expression is the same on both graphs. However, the $\langle a \neq \varepsilon \rangle$ differentiates the two. The proof for $\sim_=$ and $\sim$ is similar.

To see that with the presence of path negation the $\sim_=$ fragment can define $a_\neq$ observe that $\alpha_\neq$ is equivalent to $\overline{\alpha_=} \cap \alpha$.

Also, Proposition 7.3.14 and the discussion before Theorem 7.4.1 implies that $\mathsf{GXPath}_{\text{core}}(\mathsf{eq}_=)$ is strictly contained in $\mathsf{GXPath}_{\text{core}}(\sim_=)$. □

Note that some of the inclusions in Theorem 7.4.2 are not proved to be strict. We do however conjecture that all of the unmarked inclusions are indeed strict.

7.5 Conjunctive Graph XPath queries

In order to obtain a more practical language one often defines a class of conjunctive queries based on a well selected set of primitives [Abiteboul et al., 1995]. Here we define the class of *conjunctive GXPath* queries and analyse query evaluation bounds induced by this extension. In particular we show that the complexity is the best possible in light of CRPQs.

Conjunctive GXPath queries are defined as expression of the form:

$$Ans(\bar{z}) \;:=\; \bigwedge_{1\leq i\leq m} \alpha_i(x_i,y_i) \wedge \bigwedge_{1\leq j\leq m'} \psi_j(x_j), \tag{7.1}$$

where $m, m' > 0$, each α_i is a path expression, each ψ_j a node expression, and $\bar{z}$ is a tuple of variables among $\bar{x}$ and $\bar{y}$. A query with the head $Ans()$ (i.e., no variables in the output) is called a *Boolean* query.

These queries extend their base atoms with conjunction, as well as existential quantification: variables that appear in the body but not in the head (i.e., variables in $\bar{x}$ and $\bar{y}$ but not $\bar{z}$) are assumed to be existentially quantified.

The semantics of a conjunctive GXPath query Q of the form (7.1) over a data graph $G = \langle V, E, \rho \rangle$ is defined as follows. Given a valuation $\nu : \bigcup_{1\leq i\leq m}\{x_i,y_i\} \cup \bigcup_{1\leq j\leq m'}\{x_j\} \to V$, we write $(G,\nu) \models Q$ if $(\nu(x_i),\nu(y_i))$ is in $[\![\alpha_i]\!]^G$, for each $i = 1,\ldots,m$ and $\nu(x_j) \in [\![\psi_j]\!]^G$, for $j = 1,\ldots,m'$. Then $Q(G)$ is defined as the set of all tuples $\nu(\bar{z})$ such that $(G,\nu) \models Q$. If Q is Boolean, we let $Q(G)$ be true if $(G,\nu) \models Q$ for some ν (that is, as usual, the empty tuple models the Boolean constant true, and the empty set models the Boolean constant false).

Example 7.5.1. *Coming back to the example with actors and movies or documentaries they appear in (Figure 2.3), we can now ask for people who have collaborated both with Kevin Bacon and Paul Erdős. This query is defined by:*

$$Q(x) = (x, (\mathit{cast}^- \cdot \mathit{cast})^*[= \mathit{Kevin\ Bacon}], y) \wedge (x, (\mathit{cast}^- \cdot \mathit{cast})^*[= \mathit{Paul\ Erdős}], z).$$

Note that this query is expressible by GXPath with no conjunction (by using intersection), however, the syntax used by conjunctive queries is more intuitive, especially when one needs conjunction of three or more conditions. As we show in Section 9.2, conjunction of four conditions is no longer expressible in the base language.

If the database is further extended to include people who have co-written papers, we could also express query returning people with a finite Erdős-Bacon number. For this the second conjunct in the query Q would simply change to $(x, (\mathsf{author}^- \cdot \mathsf{author})^*[= \mathsf{Erdős}], z)$*, where an author edge connects each paper with one of its authors.*

As before, we study data and combined complexity of the query evaluation problem, i.e. checking, for a query Q, a data graph G and a tuple of nodes $\overline{v}$, whether $\overline{v} \in Q(G)$ (for data complexity the query Q is fixed).

Theorem 7.5.2.
- *Data complexity for conjunctive GXPath queries is in* PTIME.
- *Combined complexity is* NP*-complete.*

The data complexity bound easily follows from query evaluation bounds for GXPath queries. For combined complexity we do the standard guess and check algorithms, using again the fact that the language can be evaluated in PTIME. The NP lower bound follows from the result for CRPQs [Barceló et al., 2012b].

7.6 Summary

As we have seen in this chapter there are many flavours and variants of GXPath, defined by the set of navigational properties or data value tests they use. Studying them leads to a conclusion that all of them posses several desirable properties. Namely, query evaluation is always in PTIME, and several linear-time fragments can be isolated. Furthermore adding conjunction does not increase the complexity above that for CRPQs – the simplest class of conjunctive queries over graphs. Another desirable property is the simplicity of use. Indeed, we have seen through several examples that many interesting queries can be expressed in a clear and succinct manner, avoiding cumbersome constructions such as the ones used in register automata or the related classes of regular-like expressions. In the end we have also identified several subclasses capturing natural FO fragments. From all of this we can conclude that GXPath forms a good basis for graph query languages and in particular, some fragments should be considerer as the logical core for any such language. To be more precise, we believe that the following two fragments should be considered as basic primitives when designing a graph language:

- $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}(\mathsf{c})$ – This language was shown to have linear time evaluation and still retains a reasonable amount of expressive power. One of the negative sides is the inability

to capture negation, thus making it strictly weaker than FO^3, however, the navigational part is essentially PDL and therefore firmly rooted in logic.

- $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \mathsf{eq}, \sim)$ – While the complexity of evaluation here jumps to cubic, we can restore the connection with FO enriched with data tests and binary transitive closure. Therefore, we strongly believe that this language, or some of its variants, should be considered as the logical kernel of any query language for graphs.

Chapter 8

Beyond graphs – TriAL

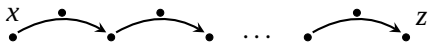
The Semantic Web and its underlying data model, RDF, are usually cited as one of the key applications of graph databases, but there is some mismatch between them. Recall that the standard model of graph databases [Angles and Gutierrez, 2008, Wood, 2012] that dates back to [Consens and Mendelzon, 1990, Cruz et al., 1987], is that of directed edge-labelled graphs, i.e., pairs $G = (V, E)$, where V is a set of vertices (objects), and E is a set of labelled edges. Each labelled edge is of the form (v, a, v'), where v, v' are nodes in V, and a is a label from some finite labelling alphabet Σ. As such, they are the same as labelled transition systems used as a basic model in both hardware and software verification. Graph databases, as we have seen previously, can also store data associated with their nodes (e.g., information about each person in a social network).

The model of RDF data is very similar, yet slightly different. The basic concept is a *triple* (s, p, o), that consists of the subject s, the predicate p, and the object o, drawn from a domain of uniform resource identifiers (URI's). Thus, the middle element need not come from a finite alphabet, and may in addition play the role of a subject or an object in another triple. For instance, $\{(s, p, o), (p, s, o')\}$ is a valid set of RDF triples, but in graph databases, it is impossible to have two such edges.

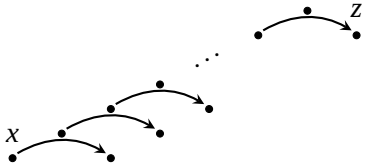
To understand why this mismatch is a problem, consider querying graph data. Since graph databases and RDF are represented as relations, relational queries can be applied to them. But crucially, we may also query the *topology* of a graph. For instance, many graph query languages have, as their basic building block, *regular path queries*, or RPQs [Cruz et al., 1987], that find nodes reachable by a path whose label belongs to a regular language.

We take the notion of reachability for granted in graph databases, but what is the corresponding notion for triples, where the middle element can serve as the source and the target of an edge? Then there are multiple possibilities, two of which are illustrated below.

Query $\mathsf{Reach}_{\rightarrow}$ looks for pairs (x,z) connected by paths of the following shape:



and $\mathsf{Reach}_{\nearrow}$ looks for the following connection pattern:



But can such patterns be defined by existing RDF query languages? Or can they be defined by existing graph query languages under some graph encoding of RDF?

To answer these questions, we need to understand which navigational facilities are available for RDF data. A recent survey of graph database systems [Angles, 2012] shows that, by and large, they either offer support for triples, or they do graphs and then can express proper reachability queries. An attempt to add navigation to RDF languages was made in [Pérez et al., 2010], where a language called nSPARQL was defined by taking SPARQL [Harris and Seaborne, 2013, Pérez et al., 2009], the standard query language for RDF, and extending it with a navigational mechanism provided by *nested regular expressions*. The evaluation of those queries uses essentially a graph encoding of RDF. As the starting point of our investigation, we show that there are natural reachability patterns for triples, similar to those shown above, that *cannot* be defined in graph encodings of RDF [Arenas and Pérez, 2011] using nested regular expressions, nor in nSPARQL itself.

Thus, navigational patterns over triples are beyond reach of both RDF languages and graph query languages that work on encodings of RDF. The solution is then to design languages that work directly on RDF triples, and have both relational and navigational querying facilities, just like graph query languages. Our goal, therefore, is to adapt graph database techniques for direct RDF querying.

A crucial property of a query language is *closure*: queries should return objects of the same kind as their input. Closed languages, therefore, are compositional: their operators can be applied to results of queries. Using graph languages for RDF suffers from non-compositionality: for instance, RPQs return graphs rather than triples. So we start by defining a closed language for triples. To understand its basic operations, we first look at a language that has essentially first-order expressivity, and then add navigational features.

We take relational algebra as the basic language. Clearly projection violates closure so we throw it away. Selection and set operations, on the other hand, are fine. The problematic operation is Cartesian product: if T, T' are sets of triples, then $T \times T'$ is not a set of triples but rather a set of 6-tuples. What do we do then? We shall need reachability in the language, and for graphs, reachability is computed by iterating *composition* of relations. The composition

operation for binary relations preserves closure: a pair (x,y) is in the composition $R \circ R'$ of R and R' iff $(x,z) \in R$ and $(z,y) \in R'$ for some z. So this is a join of R and R' and it seems that what we need is it analogue for triples.

But queries $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ demonstrate that there is no such thing as *the reachability* for triples. In fact, we shall see that there is not even a nice analogue of composition for triples. So instead, we add *all* possible joins that keep the algebra closed. The resulting language is called *Triple Algebra*, denoted by TriAL. We then add an iteration mechanism to it, to enable it to express reachability queries based on different joins, and obtain *Recursive Triple Algebra* TriAL*.

The algebra TriAL* can express both reachability patterns above, as well as queries we prove to be inexpressible in nSPARQL. It has a declarative language associated with it, a fragment of Datalog. It has good query evaluation bounds: combined complexity is (low-degree) polynomial. Moreover, we exhibit a fragment with complexity of the order $O(|e| \cdot |O| \cdot |T|)$, where e is the query, O is the set of objects in the database, and T is the set of triples. This is a very natural fragment, as it restricts arbitrary recursive definitions to those essentially defining reachability properties.

The model we use is slightly more general than just triples of objects and amounts to combining triplestores as in, e.g., [Jena, 2012] with the representation of objects used in the Neo4j database [Cudré-Mauroux and Elnikety, 2011, Neo4j, 2013]. Each object participating in a triple comes associated with a set of attributes. Attribute values are naturally drawn from an infinite alphabet, thus following the usual approach of graphs with data. Of course this can be modelled via more triples, but the model we use is conceptually cleaner and leads to a more natural comparison with standard relational languages. In particular, we show that TriAL lives between FO^3 and FO^6 (recall that FO^k refers to the fragment of First-Order Logic using only k variables). In fact it contains FO^3, is contained in FO^6, and is incomparable with FO^4 and FO^5. A similar result holds for TriAL* and transitive closure logic.

It is also worthwhile mentioning that adding data values to RDF triplestores leads to a more natural representation of data, allowing us to describe a certain resource by its set of attributes. This property also makes it easy to represent data graphs as RDF documents, allowing for data values in either nodes or edges (or both). We will return to this when comparing TriAL* to graph languages in Chapter 9.

8.1 Graph databases and RDF

RDF databases RDF databases contain triples in which, unlike in graph databases, the middle component need not come from a fixed set of labels. Formally, if $\mathbf{U}$ is a countably infinite domain of uniform resource identifiers (URI's), then an RDF triple is $(s,p,o) \in \mathbf{U} \times \mathbf{U} \times \mathbf{U}$,

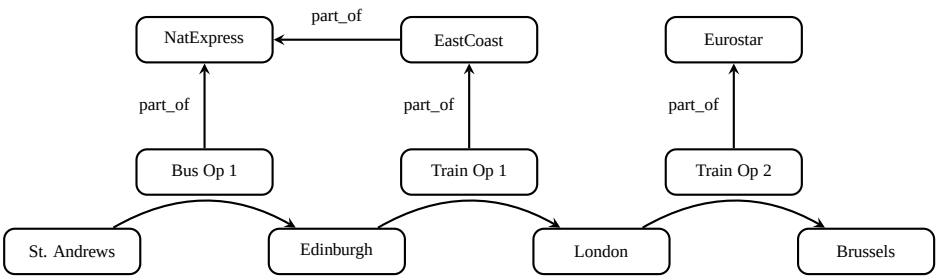


Figure 8.1: RDF graph storing information about cities and transport services between them

where *s* is referred to as the subject, *p* as the predicate, and *o* as the object. An RDF graph is just a collection of RDF triples. Here we deal with *ground* RDF documents [Pérez et al., 2010], i.e., we do not consider blank nodes or literals in RDF documents (otherwise we need to deal with disjoint domains, which complicates the presentation).

Example 8.1.1. The RDF database *D* in Figure 8.1 contains information about cities, modes of transportation between them, and operators of those services. Each triple is represented by an arrow from the subject to the object, with the arrow itself labeled with the predicate. Examples of triples in *D* are (`Edinburgh`, `Train Op 1`, `London`) and (`Train Op 1`, `part_of`, `EastCoast`). For simplicity, we assume from now on that we can determine implicitly whether an object is a city or an operator. This can of course be modeled by adding an additional outgoing edge labeled `city` from each city and `operator` from each service operator.

Graph Queries for RDF Navigational properties (e.g., reachability patterns) are among the most important functionalities of RDF query languages. However, typical RDF query languages, such as SPARQL, are in spirit relational languages. To extend them with navigation, as in [Pérez et al., 2010, Anyanwu and Sheth, 2003, Losemann and Martens, 2012], one typically uses features inspired by graph query languages. Nonetheless, such approaches have their inherent limitations, as we explain here.

Looking again at the database *D* in Figure 8.1, we see the main difference between graphs and RDF: the majority of the edge labels in *D* are also used as subjects or objects (i.e., nodes) of other triples of *D*. For instance, one can travel from Edinburgh to London by using a train service Train Op 1, but in this case the label itself is viewed as a node when we express the fact that this operator is actually a part of EastCoast trains.

For RDF, one normally uses a model of *triplestores* that is different from graph databases. According to it, the database from Figure 8.1 is viewed as a ternary relation:

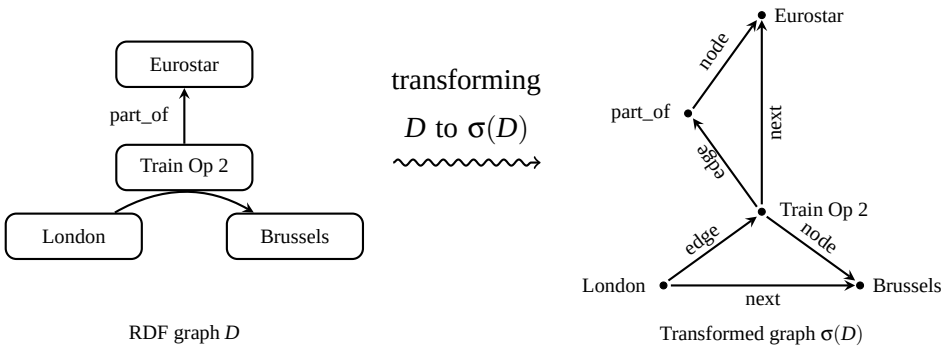


Figure 8.2: Transforming part of the RDF database from Figure 8.1 into a graph database

St. Andrews	Bus Op 1	Edinburgh
Edinburgh	Train Op 1	London
London	Train Op 2	Brussels
Bus Op 1	part_of	NatExpress
Train Op 1	part_of	EastCoast
Train Op 2	part_of	Eurostar
EastCoast	part_of	NatExpress

Suppose one wants to answer the following query:

Q: *Find pairs of cities (x,y) such that one can travel from x to y using services operated by the same company.*

A query like this is likely to be relevant, for instance, when integrating numerous transport services into a single ticketing interface. In our example, the pair $(\texttt{Edinburgh},\texttt{London})$ belongs to $Q(D)$, and one can also check that $(\texttt{St. Andrews},\texttt{London})$ is in $Q(D)$, since recursively both operators are part of NatExpress (using the transitivity of part_of). However, the pair $(\texttt{St. Andrews},\texttt{Brussels})$ does not belong to $Q(D)$, since we can only travel that route if we change companies, from NatExpress to Eurostar.

To enhance SPARQL with navigational properties, [Pérez et al., 2010] added nested regular expressions to it, resulting in a language called nSPARQL. The idea was to combine the usual reachability patterns of graph query languages with the XPath mechanism of node tests. However, nested regular expressions, which we saw earlier, are defined for graphs, and not for databases storing triples. Thus, they cannot be used directly over RDF databases; instead, one needs to transform an RDF database D into a graph first. An example of such transformation $D \to \sigma(D)$ was given in [Arenas and Pérez, 2011]; it is illustrated in Figure 8.2.

Formally, given an RDF document D, the graph $\sigma(D) = (V,E)$ is a graph database over alphabet $\Sigma = \{\text{next},\text{node},\text{edge}\}$, where V contains all resources from D, and for each triple

(s,p,o) in D, the edge relation E contains edges (s,edge,p), (p,node,o) and (s,next,o). This transformation scheme is important in practical RDF applications (it was shown to be crucial for addressing the problem of interpreting RDFS features within SPARQL [Pérez et al., 2010]). At the same time, it is not sufficient for expressing simple reachability patterns like those in query Q:

Proposition 8.1.2. *The query Q is not expressible by NREs over graph transformations* $\sigma(\cdot)$ *of ternary relations.*

Proof. Consider the RDF documents D_1 and D_2 consisting of the following triples:

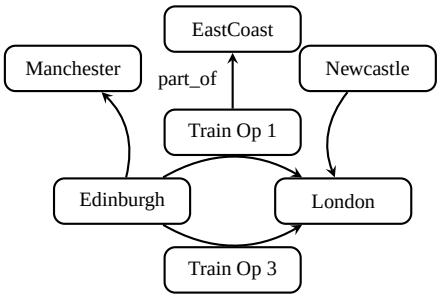
Graph D_1:

St Andrews	Bus Operator 1	Edinburgh
Edinburgh	Train Op 1	London
Edinburgh	Train Op 3	London
Edinburgh	Train Op 1	Manchester
Newcastle	Train Op 1	London
London	Train Op 2	Brussels
Bus Operator 1	part of	NatExpress
Train Op 1	part of	EastCoast
Train Op 2	part of	Eurostar
EastCoast	part of	NatExpress

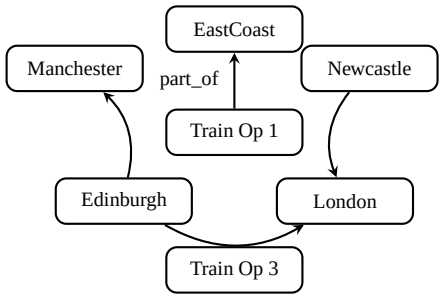
Graph D_2:

St Andrews	Bus Operator 1	Edinburgh
Edinburgh	Train Op 3	London
Edinburgh	Train Op 1	Manchester
Newcastle	Train Op 1	London
London	Train Op 2	Brussels
Bus Operator 1	part of	NatExpress
Train Op 1	part of	EastCoast
Train Op 2	part of	Eurostar
EastCoast	part of	NatExpress

Essentially, graph D_1 is an extension of the RDF document D in Figure 8.1, while graph D_2 is the same as D_1 except that it does not contain the triple (Edinburgh, Train Op 1, London). The relevant parts of our databases are illustrated in the following image.



Part of RDF graph D_1



Part of RDF graph D_2

The absence of this triple has severe implications with respect to the query Q of the statement of the Proposition, since in particular the pair (St Andrews, London) belongs to the evaluation of Q over D_1, but it does not belong to the evaluation of Q over D_2.

However, it is not difficult to check that the graph translations of D_1 and D_2 are exactly the same graph database: $\sigma(D_1) = \sigma(D_2)$. We have included the relevant part of transformations

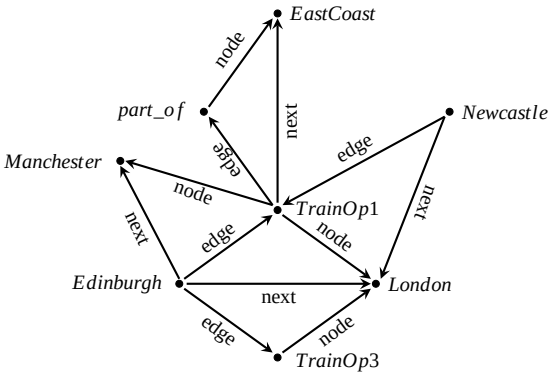


Figure 8.3: Transforming part of the RDF databases D_1 and D_2

$\sigma(D_1)$ and $\sigma(D_2)$ in Figure 8.3. It follows that Q is not expressible in nested regular expressions, since obviously the answer of all nested regular expressions is the same over $\sigma(D_1)$ and $\sigma(D_2)$ (they are the same graph).

□

Thus, the most common RDF navigational mechanism cannot express a very natural property, essentially due to the need to do so via a graph transformation.

One might argue that this result is due to the shortcomings of a specific transformation (however relevant to practical tasks it might be). So we ask what happens in the native RDF scenario. In particular, we would like to see what happens with the language nSPARQL [Pérez et al., 2010], which is a proper RDF query language extending SPARQL with navigation based on nested regular expressions. But this language falls short too, as it fails to express the simple reachability query Q.

Theorem 8.1.3. *The query Q above cannot be expressed in nSPARQL.*

Proof. The semantics of the nested regular expressions in the RDF context (in [Pérez et al., 2010]) is given as follows, assuming a triple representation of RDF documents. For next, it is the set $\{(v,v') \mid \exists z E(v,z,v')\}$, the semantics of edge is $\{(v,v') \mid \exists z E(v,v',z)\}$ and node is $\{(v,v') \mid \exists z E(z,v,v')\}$; for the rest of the operators it is the same as in the graph database case. Thus, even though stated in an RDF context, this semantics is essentially given according to the translation $\sigma(\cdot)$, in the sense that the semantics of an NRE e is the same for all RDF documents D and D' such that $\sigma(D) = \sigma(D')$ [1]. Hence the proof follows directly from Proposition 8.1.2 and the easy fact that Q cannot be expressed in SPARQL. □

[1] The NREs defined in [Pérez et al., 2010] had additional primitives, such as next :: sp. These were added for the purpose of allowing RDFS inference with NREs, but play no role in the general expressivity of nSPARQL in our setting since we are dealing with arbitrary objects, whereas the constructs in [Pérez et al., 2010] are limited to RDFS predicates. Here we assume that primitives such as next :: [e], with e an arbitrary NRE, are not allowed. For a discussion on how the proof extends in the case when they are present see [Pérez et al., 2010]

The key reason for these limitations is that the navigation mechanisms used in RDF languages are graph-based, when one really needs them to be triple-based.

Triplestore Databases To introduce proper triple-based navigational languages, we first define a simple model of triplestores. Let O be a countably infinite set of objects, and $\mathcal{D}$ be a countably infinite set of data values.

Definition 8.1.4. *A* triplestore database*, or just* triplestore *over $\mathcal{D}$ is a tuple $T = (O, E_1, \ldots, E_n, \rho)$, where:*

- *$O \subset O$ is a finite set of objects,*
- *each $E_i \subseteq O \times O \times O$ is a set of triples, and*
- *$\rho : O \to \mathcal{D}$ is a function that assigns a data value to each object.*

Often we have just a single ternary relation E in a triplestore database (e.g., in the previously seen examples of representing RDF databases), but all the languages and results we state here apply to multiple relations. The function ρ could also map O to tuples over $\mathcal{D}$, and all results remain true (one just uses $\mathcal{D}^k$ as the range of ρ, as in the example below). We use the function $\rho : O \to \mathcal{D}$ just to simplify notations.

Triplestores easily model RDF, and we will see later that they model data graphs. To further illustrate the usefulness of adding data values to triples, we now show how they can be used to model social networks. Consider a scenario where each user has a set of attributes attached to her/his entity (in our example, name, email, and age). Values of attributes come from an infinite domain of data values, while each user is uniquely described by the id value describing one object in the model. Users form connections, also labelled with data (e.g., creation date and type of the connection). Note that such social networks could simply be viewed as graph databases with multiple attributes and values attached both to edges and to the nodes (see Section 2.1). A part of this network is presented in Figure 8.4.

In the triplestore representation of this network, O is the set of all user and connection ids, while the data value function assigns to each object in O a quintuple *(name,email,dob,type,time)* of values, each with the natural domain. We use quintuples to represent data values and assume that each user entity will have null values for the last two attributes, while a connection entity will have nulls in the first three. Another way to go around this would be to have two different data value assignments to the object attributes, one for user objects and another for connection objects. To keep our language one sorted and compact we opt for the option presented here. The triples thus are

o175	c163	o122
o175	c137	o7521
o7521	c177	o122

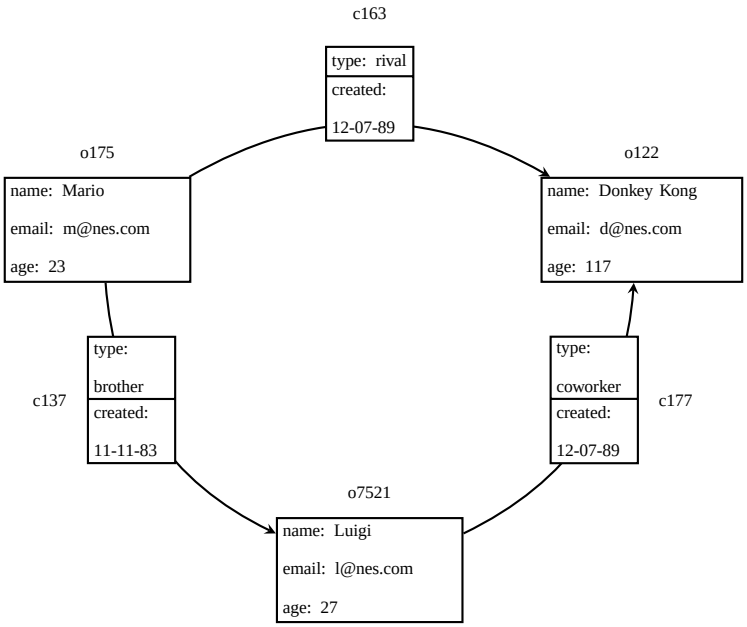


Figure 8.4: A social network graph

and the data values assignments function ρ is:

$\rho(o175)$ = (Mario,m@nes.com,23,$\perp,\perp$)

$\rho(o122)$ = (Donkey Kong,d@nes.com,117,$\perp,\perp$)

$\rho(o7521)$ = (Luigi,l@nes.com,27,$\perp,\perp$)

$\rho(c137)$ = ($\perp,\perp,\perp$,brother,11–11–83)

$\rho(c177)$ = ($\perp,\perp,\perp$,coworker,12–07–89)

$\rho(c163)$ = ($\perp,\perp,\perp$,rival,12–07–89)

Thus, triplestores describe a simple data model that is applicable in a wide range of scenarios, including RDF, graph databases and social networks.

8.2 An Algebra for RDF

We saw that problems encountered while adapting graph languages to RDF are related to the inherent limitations of the graph data model for representing RDF data. Thus, one should work directly with triples. But existing languages are either based on binary relations and fall short of the power necessary for RDF querying, or are general relational languages which are not closed when it comes to querying RDF triples. Hence, we need a language that works directly on triples, is closed, and has good query evaluation properties.

We now present such a language, based on relational algebra for triples. We start with a plain version and then add recursive primitives that provide the crucial functionality for handling reachability properties.

The operations of the usual relational algebra are selection, projection, union, difference, and cartesian product. Our language must remain *closed*, i.e., the result of each operation ought to be a valid triplestore. This clearly rules out projection. Selection and Boolean operations are fine. Cartesian product, however, would create a relation of arity six, but instead we use *joins* that only keep three positions in the result.

Triple joins To see what kind of joins we need, let us first look at the *composition* of two relations. For binary relations S and S', their composition $S \circ S'$ has all pairs (x,y) so that $(x,z) \in S$ and $(z,y) \in S'$ for some z. Reachability with relation S is defined by recursively applying composition: $S \cup S \circ S \cup S \circ S \circ S \cup \ldots$. So we need an analog of composition for triples. To understand how it may look, we can view $S \circ S'$ as the *join* of S and S' on the condition that the 2nd component of S equals the first of S', and the output consist of the remaining components. We can write it as

$$S \underset{2=1'}{\overset{1,2'}{\bowtie}} S'$$

Here we refer to the positions in S as 1 and 2, and to the positions in S' as $1'$ and $2'$, so the join condition is $2 = 1'$ (written below the join symbol), and the output has positions 1 and $2'$. This suggests that our join operations on triples should be of the form $R \bowtie^{i,j,k}_{\text{cond}} R'$, where R and R' are tertiary relations, $i,j,k \in \{1,2,3,1',2',3'\}$, and cond is a condition (to be defined precisely later).

But what is the most natural analog of relational composition? Note that to keep three indexes among $\{1,2,3,1',2',3'\}$, we ought to project away three, meaning that two of them will come from one argument, and one from the other. Any such join operation on triples is bound to be *asymmetric*, and thus cannot be viewed as a full analog of relational composition.

So what do we do? Our solution is to add *all* such join operations. Formally, given two tertiary relations R and R', *join* operations are of the form

$$R \underset{\theta,\eta}{\overset{i,j,k}{\bowtie}} R',$$

where

- $i,j,k \in \{1,1',2,2',3,3'\}$,
- θ is a set of equalities and inequalities between elements in $\{1,1',2,2',3,3'\} \cup O$,
- η is a set of equalities and inequalities between elements in $\{\rho(1),\rho(1'),\rho(2),\rho(2'),\rho(3),\rho(3')\} \cup \mathcal{D}$.

The semantics is defined as follows: (o_i,o_j,o_k) is in the result of the join iff there are triples $(o_1,o_2,o_3) \in R$ and $(o_{1'},o_{2'},o_{3'}) \in R'$ such that

- each condition from θ holds; that is, if $l = m$ is in θ, then $o_l = o_m$, and if $l = o$, where o is an object, is in θ, then $o_l = o$, and likewise for inequalities;
- each condition from η holds; that is, if $\rho(l) = \rho(m)$ is in η, then $\rho(o_l) = \rho(o_m)$, and if $\rho(l) = d$, where d is a data value, is in η, then $\rho(o_l) = d$, and likewise for inequalities.

Triple Algebra We now define the expressions of the *Triple Algebra*, or TriAL for short. It is a restriction of relational algebra that guarantees closure, i.e., the result of each expression is a triplestore.

- Every relation name in a triplestore is a TriAL expression.
- If e is a TriAL expression, θ a set of equalities and inequalities over $\{1,2,3\} \cup O$, and η is a set of equalities and inequalities over $\{\rho(1),\rho(2),\rho(3)\} \cup \mathcal{D}$, then $\sigma_{\theta,\eta}(e)$ is a TriAL expression.
- If e_1, e_2 are TriAL expressions, then the following are TriAL expressions:
  - $e_1 \cup e_2$;
  - $e_1 - e_2$;
  - $e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, where i, j, k, θ, η as in the definition of the join above.

The semantics of the join operation has already been defined. The semantics of the Boolean operations is the usual one. The semantics of the selection is defined in the same way as the semantics of the join (in fact, the operator itself can be defined in terms of joins): one just chooses triples (o_1, o_2, o_3) satisfying both θ and η.

Given a triplestore database T, we write $e(T)$ for the result of expression e on T.

Note that $e(T)$ is again a triplestore, and thus TriAL defines closed operations on triplestores. This is important, for instance, when we require RDF queries to produce RDF graphs as their result (instead of arbitrary tuples of objects), as it is done in SPARQL via the CONSTRUCT operator [Harris and Seaborne, 2013].

Example 8.2.1. To get some intuition about the Triple Algebra consider the following TriAL expression:

$$e = E \overset{1,3',3}{\underset{2=1'}{\bowtie}} E$$

Indexes $(1,2,3)$ refer to positions of the first triple, and indexes $(1',2',3')$ to positions of the second triple in the join. Thus, for two triples (x_1,x_2,x_3) and $(x_{1'},x_{2'},x_{3'})$, such that $x_2 = x_{1'}$, expression e outputs the triple $(x_1,x_{3'},x_3)$. E.g., in the triplestore of Fig. 8.1, (`London`, `Train Op 2`, `Brussels`) is joined with (`Train Op 2`, `part_of`, `Eurostar`), producing (`London`, `Eurostar`, `Brussels`); the full result is

St. Andrews	NatExpress	Edinburgh
Edinburgh	EastCoast	London
London	Eurostar	Brussels

Thus, e computes travel information for pairs of European cities together with companies one can use. It fails to take into account that EastCoast is a part of NatExpress. To add such information to query results (and produce triples such as (Edinburgh, NatExpress, London)), we use $e' = e \cup (e \bowtie^{1,3',3}_{2=1'} E)$.

Definable operations: intersection and complement. As usual, the intersection operation can be defined as $e_1 \cap e_2 = e_1 \bowtie^{1,2,3}_{1=1',2=2',3=3'} e_2$. Note that using join and union, we can define the set U of all triples (o_1, o_2, o_3) so that each o_i occurs in our triplestore database T. For instance, to collect all such triples so that o_1 occurs in the first position of R, and o_2, o_3 occur in the 2nd and 3rd position of R' respectively, we would use the expression $(R \bowtie^{1,2',3} R') \bowtie^{1,2,3'} R'$. Taking the union of all such expressions, gives us the relation U.

Using such U, we can define e^c, the complement of e with respect to the active domain, as $U - e$. In what follows, we regularly use intersection and complement in our examples.

Adding Recursion One problem with Example 8.2.1 above is that it does not include triples (city_1,service,city_2) so that relation R contains a triple (city_1,service_0,city_2), and there is a chain, of some length, indicating that service_0 is a part of service. The second expression in Example 8.2.1 only accounted for such paths of length 1. To deal with paths of arbitrary length, we need reachability, which relational algebra is well known to be incapable of expressing. Thus, we need to add recursion to our language.

To do so, we expand TriAL with *right* and *left Kleene closure* of any triple join $\bowtie^{i,j,k}_{\theta,\eta}$ over an expression e, denoted as $(e \bowtie^{i,j,k}_{\theta,\eta})^*$ for right, and $(\bowtie^{i,j,k}_{\theta,\eta} e)^*$ for left. These are defined as

$$\begin{aligned} (e \bowtie)^* &= \emptyset \cup e \cup e \bowtie e \cup (e \bowtie e) \bowtie e \cup \ldots, \\ (\bowtie e)^* &= \emptyset \cup e \cup e \bowtie e \cup e \bowtie (e \bowtie e) \cup \ldots \end{aligned}$$

We refer to the resulting algebra as *Triple Algebra with Recursion* and denote it by TriAL^*.

When dealing with binary relations we do not have to distinguish between left and right Kleene closures, since the composition operation for binary relations is associative. However, as the following example shows, joins over triples are not necessarily associative, which explains the need to make this distinction.

Example 8.2.2. Consider a triplestore database $T = (O, E)$, with $E = \{(a,b,c),(c,d,e),(d,e,f)\}$. The function ρ is not relevant for this example. The expression

$$e_1 = (E \overset{1,2,2'}{\underset{3=1'}{\bowtie}})^*$$

computes $e_1(T) = E \cup \{(a,b,d),(a,b,e)\}$, while

$$e_2 = (\underset{3=1'}{\overset{1,2,2'}{\bowtie}} E)^*$$

computes $e_2(T) = E \cup \{(a,b,d)\}$.

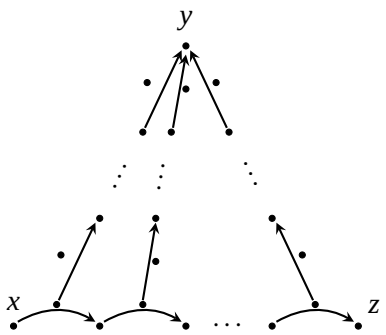
Now we present several examples of queries one can ask using the Triple Algebra.

Example 8.2.3. We refer now to reachability queries $\mathsf{Reach}_{\rightarrow}$ and $\mathsf{Reach}_{\nearrow}$ from the introduction to Chapter 8. It can easily be checked that these are defined by

$$(E \underset{3=1'}{\overset{1,2,3'}{\bowtie}})^* \quad \text{and} \quad (\underset{1=2'}{\overset{1',2',3}{\bowtie}} E)^*$$

respectively.

Next consider the query from Theorem 8.1.2. Graphically, it can be represented as follows:



That is, we are looking for pairs of cities such that one can travel from one to the other using services operated by the same company. This query is expressed by

$$((E \underset{2=1'}{\overset{1,3',3}{\bowtie}})^* \underset{3=1',2=2'}{\overset{1,2,3'}{\bowtie}})^*.$$

Note that the interior join $(E \underset{2=1'}{\overset{1,3',3}{\bowtie}})^*$ computes all triples (x,y,z), such that $E(x,w,z)$ holds for some w, and y is reachable from w using some E-path. The outer join now simply computes the transitive closure of this relation, taking into account that the service that witnesses the connection between the cities is the same.

Another useful application of such a nested query can be found in workflows tracking provenance of some document. Indeed, there we might be interested to find all versions of a document that contain an error, but originate from an error-free version. We might also ask if there is a path connecting those two documents where each of the versions referred to some particular document – the likely culprit for the mistake. In the image above z would represent version with an error, x a valid version it originates from, and y the document all of the versions that lead to the one with an error refer to.

Remark 8. *Here we give some remarks about notation and implicit assumptions in the remainder of this chapter.*

- *We will often denote conditions* θ *and* η *as conjunction of equalities or inequalities instead of sets. For example we will write* $\theta = (1 \neq 3') \wedge (2 = 2')$ *for* $\theta = \{1 \neq 3', 2 = 2'\}$.
- *In the proofs we will usually handle only the case of the right Kleene closure* $(R\bowtie)^*$. *The proofs for the left closure are completely symmetric.*
- *As usual in database theory, we only consider queries that are domain-independent, and therefore we loose no generality in assuming active domain semantics for FO formulas and other similar formalisms.*

8.3 A Declarative Language

Triple Algebra and its recursive versions are *procedural* languages. In databases, we are used to dealing with declarative languages. The most common one for expressing queries that need recursion is Datalog. It is one of the most studied database query languages, and it has reappeared recently in numerous applications. One instance of this is its well documented success in Web information extraction [Gottlob and Koch, 2004] and there are numerous others. So it seems natural to look for Datalog fragments to capture TriAL and its recursive version.

Since Datalog works over relational vocabularies, we need to explain how to represent triplestores T. The schema of these representations consists of a ternary relation symbol $E(\cdot,\cdot,\cdot)$ for each triplestore name in T, plus a binary relation symbol $\sim(\cdot,\cdot)$. Each triplestore database T can be represented as an instance I_T of this schema in the standard way: the interpretation of each relation name E in this instance corresponds to the triples in the triplestore E in T, and the interpretation of $\sim$ contains all pairs (x,y) of objects such that $\rho(x) = \rho(y)$, i.e. x and y have the same data value. If the values of ρ are tuples, we just use $\sim_i$ relations testing that the ith components of tuples are the same, for each i; this does not affect the results presented below.

We start with a Datalog fragment capturing TriAL. A TripleDatalog rule is of the form

$$S(\overline{x}) \leftarrow S_1(\overline{x_1}), S_2(\overline{x_2}), \sim(y_1,z_1), \ldots, \sim(y_n,z_n), u_1 = v_1, \ldots, u_m = v_m \tag{8.1}$$

where

1. S, S_1 and S_2 are (not necessarily distinct) predicate symbols of arity at most 3;
2. all variables in $\overline{x}$ and each of y_i, z_i and u_j, v_j are contained in $\overline{x_1}$ or $\overline{x_2}$.

A TripleDatalog$^\neg$ rule is like the rule (8.1) but all equalities and predicates, except the head predicate S, can appear negated. A TripleDatalog$^\neg$ *program* Π is a finite set of TripleDatalog$^\neg$

rules. Such a program Π is *non-recursive* if there is an ordering $r_1,\ldots,r_k$ of the rules of Π so that the relation in the head of r_i does not occur in the body of any of the rules r_j, with $j \leq i$.

As is common with non-recursive programs, the semantics of nonrecursive TripleDatalog$^\neg$ programs is given by evaluating each of the rules of Π, according to the order $r_1,\ldots,r_k$ of its rules, and taking unions whenever two rules have the same relation in their head (see [Abiteboul et al., 1995] for the precise definition). We are now ready to present the first capturing result.

Proposition 8.3.1. TriAL *is equivalent to nonrecursive* TripleDatalog$^\neg$ *programs.*

Proof. Let us first show the containment of TriAL in non-recursive TripleDatalog$^\neg$. We show that for every expression e one can construct a non-recursive TripleDatalog$^\neg$ program Π_e such that, $e(T) = \Pi_e(I_T)$, for all triplestore databases T.

We define the translation by the following inductive construction, assuming Ans, Ans_1 and Ans_2 are special symbols that define the output of non-recursive TripleDatalog$^\neg$ programs.

- If e is just a triplestore name E, then Π_e consists of the single rule $Ans(x,y,z) \leftarrow E(x,y,z)$.
- If e is $e_1 \cup e_2$, then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rules $Ans(\overline{x}) \leftarrow Ans_1(\overline{x})$ and $Ans(\overline{x}) \leftarrow Ans_2(\overline{x})$, where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
- If e is $e_1 - e_2$, then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rule $Ans(\overline{x}) \leftarrow Ans_1(\overline{x}), \neg Ans_2(\overline{x})$, where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
- If e is $e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, assume that θ consists of m conditions, and η consists of n conditions. Then Π_e consists of the union of the rules of the programs Π_{e_1} and Π_{e_2}, together with the rule

$$Ans(x_i,x_j,x_k) \leftarrow Ans_1(x_1,x_2,x_3), Ans_2(x_4,x_5,x_6), V(y_1,z_1),\ldots,V(y_n,z_n), \\ u_1(=) \neq v_1,\ldots,u_m(=) \neq v_m, \quad (8.2)$$

  where for each p-th condition in θ of form $a = b$ or $a \neq b$, we have that $u_p = x_a$ and $v_p = x_b$ (or $u_p = o$ if a is an object o in O, and likewise for b), and for each p-th condition in θ of form $\rho(a) = \rho(b)$ or $\rho(a) \neq \rho(b)$, we have that $y_p = x_a$ and $z_p = x_b$, and V is either $\sim$ or $\neg\sim$; and where we assume that Ans_1 and Ans_2 are the predicates that define the output of Π_{e_1} and Π_{e_2}, respectively.
- The case of selection goes along the same lines as the join case.

Clearly, this program is nonrecursive. Moreover, it is trivial to prove that this transition satisfies our desired property.

Next we show the containment of non-recursive TripleDatalog$^\neg$ in TriAL. We show that for every non-recursive TripleDatalog$^\neg$ program Π one can construct an expression e_Π such that, $e_\Pi(T) = \Pi(I_T)$, for all triplestore databases T.

We assume that Π contains a single predicate *Ans* that represents the answer of the query. Also, without loss of generality we can assume that no rule uses predicate E, for some triplestore name E, other than a rule of form $P(x,y,z) \leftarrow E(x,y,z)$, for a predicate P in the predicates of Π that does not appear in the head of any other rule in Π.

We need some notation. The dependence graph of Π is a directed graph whose nodes are the predicates of π, and the edges capture the dependence relation of the predicates of Π, i.e., there is an edge from predicate R to predicate S if there is a rule in Π with R in its head and S in its body. Since Π is non-recursive, its dependency graph is acyclic. We now define the TriAL expression in a recursive fashion, following its dependency graph:

- Assume that all the rules in Π that have predicate S in the head are of form

$$S(x_{a^j}, x_{b^j}, x_{c^j}) \leftarrow S_1^j(x_1^j, x_2^j, x_3^j), S_2^j(x_4^j, x_5^j, x_6^j), (\neg)\sim(y_1^j, z_1^j), \ldots, (\neg)\sim(y_n^j, z_n^j), \\ u_1^j(\neq) = v_1^j, \ldots, u_m^j(\neq) = v_m^j \quad (8.3)$$

  for $1 \leq j \leq m$, and where S_1^j and S_2^j are (not necessarily distinct) predicate symbols of arity at most 3 and all variables in $x_{a^j}, x_{b^j}, x_{c^j}$ and each of y_i^j, z_i^j and u_k^j, v_k^j are contained in $\{x_1^j, x_2^j, x_3^j, x_4^j, x_5^j, x_6^j\}$.

  Then the TriAL expression e_S is

$$\bigcup_{1 \leq j \leq m} e_{S_1^j} \bowtie^{a^j, b^j, c^j}_{\theta^j, \eta^j} e_{S_2^j},$$

  where θ contains an (in)equality $a = b$ for each (in)equality $x_a = x_b$ in the rule, and η^j contains an (in)equality $\rho(a) = \rho(b)$ for each predicate $\sim(a,b)$ (or its negation) in the rule. If either of S_1^j or S_2^j appear negated in the rule, then just replace $e_{S_1^j}$ for $(e_{S_1^j})^c$ or $(e_{S_2^j})^c$.

- The TriAL expression e_P (for predicate P in rule $P(x,y,z) \leftarrow E(x,y,z)$) is just E; if these variables appear in different order in the rule, we permute them via the selection operator σ.

It is now straightforward to verify that for every non-recursive TripleDatalog$^\neg$ program Π whose answer predicate is *Ans* the expression e_{Ans} is such that, $e_{Ans}(T) = \Pi(I_T)$, for all triplestore databases T. □

We next turn to the expressive power of recursive Triple Algebra TriAL*. To capture it, we of course add recursion to Datalog rules, and impose a restriction that was previously used

in [Consens and Mendelzon, 1990]. A ReachTripleDatalog$^\neg$ *program* is a TripleDatalog$^\neg$ program in which each recursive predicate S is the head of exactly two rules of the form:

$$\begin{array}{lcl} S(\bar{x}) & \leftarrow & R(\bar{x}) \\ S(\bar{x}) & \leftarrow & S(\bar{x_1}), R(\bar{x_2}), V(y_1,z_1), \ldots, V(y_k,z_k) \end{array} \tag{8.4}$$

where each $V(y_i,z_i)$ is one of the following: $y_i = z_i$, or $y_i \neq z_i$, or $\sim(y_i,z_i)$, or $\neg\sim(y_i,z_i)$, and R is a nonrecursive predicate of arity at most 3, or a recursive predicate defined by a rule of the form 8.4 that appears before S. These rules essentially mimic the standard reachability rules (for binary relation) in Datalog, and in addition one can impose equality and inequality constraints, as well as data equality and inequality constraints, along the paths.

Note that the negation in ReachTripleDatalog$^\neg$ programs is *stratified*. The semantics of these programs is the standard least-fixpoint semantics [Abiteboul et al., 1995]. A similarly defined syntactic class, but over graph databases, rather than triplestores, was shown to capture the expressive power of FO with the transitive closure operator [Consens and Mendelzon, 1990]. In our case, we have a capturing result for TriAL*.

Theorem 8.3.2. *The expressive power of* TriAL* *and* ReachTripleDatalog$^\neg$ *programs is the same.*

Proof. Let us first show the containment of TriAL* in ReachTripleDatalog$^\neg$. The proof goes along the same lines as the proof of containment of TriAL in TripleDatalog$^\neg$. We have to show that for every TriAL* expression e there is a ReachTripleDatalog$^\neg$ program Π_e such that $e(T) = \Pi_e(I_T)$, for all triplestores T.

The only difference from the construction in the proof of TriAL in TripleDatalog$^\neg$ is the treatment of the constructs $e = (e_1 \bowtie^{i,j,k}_{\theta,\eta})^*$ and $e = (\bowtie^{i,j,k}_{\theta,\eta} e_1)^*$. For the former construct (the other one is symmetrical), assume that $\theta = (\bigwedge_{1\leq i\leq m} p_i(\neq) = q_i)$ and $\eta = (\bigwedge_{1\leq j\leq n} \rho(u_j)(\neq) = \rho(v_j))$. We let Π_e be the union of all rules of Π_{e_1}, plus rules

$$\begin{array}{rcl} Ans(x,y,z) & \leftarrow & Ans_1(x,y,z) \\ Ans(x_i,x_j,x_k) & \leftarrow & Ans(x_1,x_2,x_3), Ans_1(x_4,x_5,x_6), \\ & & (\neg)\sim(x_{p_1},x_{q_1}), \ldots, (\neg)\sim(x_{u_n},x_{v_n}), x_{p_1}(\neq) = x_{q_1}, \ldots, x_{p_m}(\neq) = x_{q_m}, \end{array}$$

where Ans_1 is the answer predicate of Π_{e_1}. Notice that we have assumed for simplicity there are no comparison with constants; these can be included in our translation the straightforward way. The proof that $e(T) = \Pi_e(I_T)$, for all triplestores T now follows easily.

The proof of containment of ReachTripleDatalog$^\neg$ in TriAL* also goes along the same lines as the proof that TripleDatalog$^\neg$ is contained in TriAL. The only difference is when creating expression e_S, for some recursive predicate S. From the properties of ReachTripleDatalog$^\neg$

programs, we know S is the head of exactly two rules of form

$$\begin{aligned} S(\overline{x}) &\leftarrow R(\overline{x}) \\ S(x_a,x_b,x_c) &\leftarrow S(x_1,x_2,x_3), R(x_4,x_5,x_6), V(y_1,z_1),\ldots,V(y_n,z_n), \\ &\qquad u_1(\neq)=v_1,\ldots,u_m(\neq)=v_m, \end{aligned}$$

1. R is a nonrecursive predicate of arity at most 3,
2. variables x_a,x_b,x_c and each of y_i, z_i and u_j, v_j are contained in $\{x_1,\ldots,x_6\}$, and
3. each $V(y_i,z_i)$ is either $\sim(y_i,z_i)$ or $\neg\sim(y_i,z_i)$

We then let e_S be $(e_R \bowtie^{a,b,c}_{\theta,\eta})^*$, where θ contains the inequality $p(\neq)=q$ for each predicate $x_p(\neq)=x_q$ in the rule above, or the respective comparison with constant if p or q belong to O, and η contains the (in)equality $\rho(p)(\neq)=\rho(q)$ for each predicate $\sim(x_p,x_q)$ (respectively, $\neg\sim(x_p,x_q)$).

Once again, it is straightforward to verify that e_{Ans} is such that, $e_{Ans}(T)=\Pi(I_T)$, for all triplestores T. □

We now give an example of a simple datalog program computing the query from Theorem 8.1.3.

Example 8.3.3. The following ReachTripleDatalog$^\neg$ program is equivalent to query Q from Theorem 8.1.3. Note that the answer is computed in the predicate Ans.

$$\begin{aligned} S(x_1,x_2,x_3) &\leftarrow E(x_1,x_2,x_3) \\ S(x_1,x'_3,x_3) &\leftarrow S(x_1,x_2,x_3), E(x_2,x'_2,x'_3) \\ \text{Ans}(x_1,x_2,x_3) &\leftarrow S(x_1,x_2,x_3) \\ \text{Ans}(x_1,x_2,x'_3) &\leftarrow \text{Ans}(x_1,x_2,x_3), S(x_3,x_2,x'_3) \end{aligned}$$

Recall that this query can be written in TriAL* as $Q=((E\bowtie^{1,3',3}_{2=1'})^*\bowtie^{1,2,3'}_{3=1',2=2'})^*$. The predicate S in the program computes the inner Kleene closure of the query, while the predicate Ans computes the outer closure.

8.4 Query Evaluation

In this section we analyze two versions of the query evaluation problems related to Triple Algebra. We start with query evaluation, redefined here for TriAL* queries.

Problem:	QUERYEVALUATION
Input:	A TriAL* expression e, a triplestore T and a tuple (x_1,x_2,x_3) of objects.
Question:	Is $(x_1,x_2,x_3)\in e(T)$?

Many graph query languages (e.g., RPQs, GXPath) have PTIME upper bounds for this problem, and the data complexity (i.e., when e is assumed to be fixed) is generally in NL (which cannot be improved, since the simplest reachability problem over graphs is already NL-hard). We now show that the same upper bounds hold for our algebra, even with recursion.

Proposition 8.4.1. *The problem* QUERYEVALUATION *is* PTIME*-complete, and in* NL *if the algebra expression e is fixed.*

Proof. The PTIME upper bound follows immediately from Theorem 8.4.2 below. PTIME-hardness follows from the fact that every FO^3 query can be expressed in TriAL (see Section 8.6) and the known result that evaluating FO^k queries is PTIME-hard already when $k = 3$ [Vardi, 1995].

For the NL upper bound, the idea is to divide the expression e into all its subexpression, corresponding to subtrees of the parsing tree of φ. Starting from the leaves until the root of the parse tree of e, one can guess the relevant triples that will be witnessing the presence of the query triple in the answer set $e(T)$.

Note that for this we only need to remember $O(|e|)$ triples – a number of fixed length. After we have guessed a triple for each node in the parse tree for e we simply check that they belong to the result of applying the subexpression defined by that node of the tree to our triplestore T. Thus to check that the desired complexity bound holds we need to show that each of the operations can be performed in NL, given any of the triples. This follows by an easy inductive argument.

For example, if $e = E_i$ is one of the initial relations in T, we simply check that the guessed triple is present in its table. Note that this can be done in NL.

This is done in an analogous way for the expressions of the form $e = e_1 \cup e_2$ and $e = e_1 - e_2$. To see that the claim also holds for joins, note that one only has to check that join conditions can be verified in NL. But this is a straightforward consequence of the observation that for conditions we use only comparisons of objects and their data values.

Finally, to see that the star operator $(R\bowtie^{i,j,k}_{\theta,\eta})^*$ can be implemented in NL we simply do a standard reachability argument for graphs. That is, since we are trying to verify that a specific triple (a,b,c) is in the answer to the star-join operator, we guess the sequence that verifies this. We begin by a single triple in R (and we can check that it is there in NL by the induction hypothesis) and guess each new triple in R, joining it with the previous one, until we have performed at most $|T|$ steps. □

Tractable evaluation (even with respect to combined complexity) is practically a must when dealing with very large and dynamic semi-structured databases. However, in order to make a case for the practical applicability of our algebra, we need to give more precise bounds for query evaluation, rather than describe complexity classes the problem belongs to. We now

show that TriAL* expressions can be evaluated in what is essentially cubic time with respect to the data. Thus, in the rest of the section we focus on the problem of actually computing the whole relation $e(T)$:

Problem:	QUERYCOMPUTATION
Input:	A TriAL* expression e and a triplestore database T.
Output:	The relation $e(T)$

We now analyze the complexity of QUERYCOMPUTATION. Following an assumption frequently made in papers on graph database query evaluation (in particular, graph pattern matching algorithms) as well as bounded variable relational languages (cf. [Fan et al., 2011, Fan et al., 2010a, Gottlob et al., 2002]), we consider an *array representation* for triplestores. That is, when representing a triplestore $T = (O, E_1, \ldots, E_m, \rho)$ with $O = \{o_1, \ldots, o_n\}$, we assume that each relation E_l is given by a three-dimensional $n \times n \times n$ matrix, so that the ijkth entry is set to 1 iff (o_i, o_j, o_k) is in E_l. Alternatively we can have a single matrix, where entries include sets of indexes of relations E_l that triples belong to. Furthermore we have a one-dimensional array of size n whose ith entry contains $\rho(o_i)$. Using this representation we obtain the following bounds.

Theorem 8.4.2. *The problem* QUERYCOMPUTATION *can be solved in time*

- $O(|e| \cdot |T|^2)$ *for* TriAL *expressions,*
- $O(|e| \cdot |T|^3)$ *for* TriAL* *expressions.*

Proof. The basic outline of the algorithm is as follows:

1. Build the parse tree for our expression.
2. Evaluate the subexpressions bottom-up.

Now to see that the algorithm meets the desired time bounds we simply have to show that each step of evaluating a subexpression can be performed in time $O(|T|^2)$.

We prove this inductively on the structure of subexpression e.

As stated previously, we assume that the objects are sorted and that the triplestore is given by its adjacency matrix T with the property that $T[i, j, k] = 1$ if and only if $(o_i, o_j, o_k) \in T$. If we are dealing with a triplestore that has more than one relation we will assume that we have access to each of the $n \times n \times n$ matrices representing E_i. In addition, to store data values we will use another array DV of size $|O|$ having $DV[i] = \rho(o_i)$, for $i = 1 \ldots n$. In the end, our algorithm computes, given an expression e and a triplestore T the matrix R_e such that $(o_i, o_j, o_k) \in e(T)$ iff $R_e[i, j, k] = 1$.

If $e = E_i$, the name of one of the initial triplestore matrices, we already have our answer, so no computation is needed.

If $e = R_1 \cup R_2$ and we are given the matrix representation of R_1 and R_2 (that is the adjacency matrix of the answer of R_i on our triplestore T) we simply compute R_e as the union of these two matrices. Note that this takes time $O(|T|)$.

If $e = R_1 \cap R_2$ we compute R_e as the intersection of these two matrices. That is, for each triple (i, j, k) we check if $R_1[i, j, k] = R_2[i, j, k] = 1$. Note that this takes time $O(|T|)$.

If $e = R_1 - R_2$ we compute R_e as the difference of the two matrices. That is for each (i, j, k) we set $R_e[i, j, k] = 1$ if and only if $R_1[i, j, k] = 1$ and $R_2[i, j, k] = 0$. The time required is $O(|T|)$.

If $e = \sigma_\varphi R_1$ and we are given the matrix for R_1 we can compute R_e in time $O(|e||T|)$ by traversing each triple (i, j, k), checking that $R_1[i, j, k] = 1$ and that the objects o_i, o_j and o_k satisfy the conditions specified by φ. Notice that each of these checks can be done in $|e|$ time using T and DV, since the number of comparisons in φ has a fixed upper bound, modulo comparison with constants. The comparison with constants can be done in time $|e|$ because we have to check (in)equality only with the constants that actually appear in e.

Finally, in the case that $e = R_1 \bowtie^{i',j',k'}_{\theta,\eta} R_2$ we can compute R_e using the following algorithm:

Procedure 1 Computing joins

Input: Matrix representation of R_1, R_2

Output: Matrix R_e representing e

1: Let θ' and η' be the conditions obtained from θ, η by removing comparisons with constants
2: Let α, β be the conditions in θ, η using constants
3: Filter R_1 and R_2 according to α, β
4: **for** $i = 1 \to n$ **do**
5: **for** $j = 1 \to n$ **do**
6: **for** $k = 1 \to n$ **do**
7: **if** $R_1[i, j, k] = 1$ **then**
8: **for** $l = 1 \to n$ **do**
9: **for** $m = 1 \to n$ **do**
10: **for** $n = 1 \to n$ **do**
11: **if** $R_2[l, m, n] = 1$ **then**
12: **if** (o_i, o_j, o_k) and (o_l, o_m, o_n) satisfy the conditions in θ', η' **then** $R_e[i', j', k'] = 1$
13: **else** $R_e[i', j', k'] = 0$

Note that lines 1–3 correspond to computing selections operator and can therefore be performed using the time $O(|e||T|)$ and reusing the matrices R_1 and R_2. It is straightforward to see

that the remaining of the algorithm works as intended by joining the desirable triples. This is performed in $O(|T|^2)$. Thus the whole join computation can be done in time $O(|T|^2)$.

This concludes the first part of our theorem and we thus conclude that TriAL query computation problem can be solved in time $O(|e||T|^2)$.

For the second part of the theorem we only have to show that each star operation can be computed in time $O(|T|^3)$. To see this we consider the following algorithm, computing the answer set for $e = (R_1 \bowtie^{i',j',k'}_{\theta,\eta})^*$

Procedure 2 Computing stars

Input: Matrix representation of R_1

Output: Matrix R_e representing e

1: Initialize $R_e := R_1$

2: **for** $i = 1 \to n^3$ **do**

3: Compute $R_e := R_e \cup R_e \bowtie^{i',j',k'}_{\theta,\eta} R_1$

First we note that the algorithm does indeed compute the correct answer set. This follows because the joining in our star process has to became saturated after n^3 steps, since this is the maximum possible number of triples in a model with n elements. Note now that each join in step 3 can be computed in time $O(|T|^2)$, thus giving us the total running time of $O(n^3 \cdot |T|^2) = O(|T|^3)$.

Finally, note that left-joins can be computed in an analogous way. □

Note that this immediately gives the PTIME upper bound for Proposition 8.4.1.

One can examine the proofs of Proposition 8.3.1 and Theorem 8.3.2 and see that translations from Datalog into algebra are linear-time. Thus, we have the same bound for the query computation problem, when we evaluate a Datalog program Π in place of an algebra expression.

Corollary 8.4.3. *The problem* QUERYCOMPUTATION *for Datalog programs* Π *can be solved in time*

- $O(|\Pi| \cdot |T|^2)$ *for* TripleDatalog$^\neg$ *programs,*
- $O(|\Pi| \cdot |T|^3)$ *for* ReachTripleDatalog$^\neg$ *programs.*

8.5 Low-complexity fragments

Even though we have acceptable combined complexity of query computation, if the size of T is very large, one may prefer to lower it even further. We now look at fragments of TriAL* for which this is possible.

Relational fragments of TriAL Ineq algorithms from Theorem 8.4.2, the main difficulty arises from the presence of inequalities in join conditions. A natural restriction then is to look at a fragment $\text{TriAL}^{=}$ of TriAL in which all conditions θ and η used in joins can only use equalities. This fragment allows us to lower the $|T|^2$ complexity, by replacing one of the $|T|$ factors by $|O|$, the number of distinct objects.

Proposition 8.5.1. *The* QUERYCOMPUTATION *problem for* $\text{TriAL}^{=}$ *expressions can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

Proof. To prove this we will use the close connection of positive fragment of $\text{TriAL}^{=}$ with FO^4. We establish this as follows. To each triplestore $T = (O, E_1, \ldots, E_n, \rho)$ we associate an FO structure $\mathcal{M}_T = (O, E_1, \ldots, E_n, \sim)$, where O is the set of objects appearing in T, $E_1, \ldots, E_n$ are just the representation of the triplestores, and $\sim(o_1, o_2)$ holds iff $\rho(o_1) = \rho(o_2)$ (they have the same data value). In Lemma 8.5.2 we will then show that for each $\text{TriAL}^{=}$ expression e one can compute, in time $O(|e|)$, an equivalent FO formula φ_e true precisely for the triples in $\mathcal{M}_T$ which satisfy e over T.

Note that we can compute $\mathcal{M}_T$ from T in linear time. To finish the proof we show in Lemma 8.5.3 that each FO^4 formula φ using relations that are at most ternary (in fact this holds for relations of arity four as well, but is not relevant for our analysis) can be evaluated in time $O(|\varphi| \cdot |O|^4)$.

The result of Proposition 8.5.1 now follows, since we can take our expression e, transform it into a formula φ_e of FO^4 and evaluate it in time $O(|\varphi_e| \cdot |O|^4) = O(|e| \cdot |O| \cdot |T|)$, since $|T| = |O|^3$ and $|\varphi_e| = O(|e|)$.

The proof of the two lemmas follows below. □

First we show that over triplestores $\text{TriAL}^{=}$ is contained in FO^4.

Lemma 8.5.2. *For every* $\text{TriAL}^{=}$ *expression* e *one can construct an* FO^4 *formula* φ_e *such that a triple* (a,b,c) *belongs to* $e(T)$ *if and only if* $\mathcal{M}_T \models \varphi_e(a,b,c)$.

Proof. The proof is done by induction. The base case when $e = E_i$ for some $1 \leq i \leq n$ is trivial, and so are the cases when $e = e_1 \cup e_2$, $e = e_1 - e_2$ and $e = \sigma_{\theta,\eta} e_1$. The only interesting case is when $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$.

As usual, we assume that e is $e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, where θ is a conjunction of equalities between elements in $\{1, 1', 2, 2', 3, 3'\} \cup O$ and η is a conjunction of equalities between elements in $\{\rho(1), \rho(1').\rho(2), \rho(2'), \rho(3), \rho(3')\}$. We need some terminology.

Let $\theta = \theta_\ell \wedge \theta_r \wedge \theta_\bowtie \wedge \theta^c_\ell \wedge \theta^c_r$, where

- θ_ℓ and θ_r contain only equalities between indexes in $\{1,2,3\}$ and $\{1',2',3'\}$, respectively.
- θ^c_ℓ and θ^c_r contain only equalities where one element is in O and the other is in $\{1,2,3\}$ and $\{1',2',3'\}$, respectively.

- $\theta_{\bowtie}$ contains all the remaining equalities, i.e. those equalities in which one index is in $\{1,2,3\}$ and the other in $\{1',2',3'\}$.

We also divide $\eta = \eta_\ell \wedge \eta_r \wedge \eta_{\bowtie}$ in the same fashion (recall that for the sake of readability we assume no comparison between data values and constants, two avoid two sorted structures). Notice that any two equalities of form $i = j'$ and $i = k'$, for $i \in \{1,2,3\}$ and $j',k' \in \{1',2',3'\}$ can be replaced with $i = j'$ and $j' = k'$, and likewise we can replace $i = k'$ and $j = k'$ with $i = j$ and $j = k'$. For this reason we assume that $\theta_{\bowtie}$ (and $\eta_{\bowtie}$) contain at most 3 equalities, and no two equalities in them can mention the same element. Furthermore, if $\theta_{\bowtie}$ has two or more equalities, then the join can be straightforwardly expressed in FO^4, since now instead of the six possible positions we only care about four -or three-of them. For this reason we only show how to construct the formula when $\theta_{\bowtie}$ has one or no equalities.

Finally, for a conjunction θ of equalities between element in $\{1,1',2,2',3,3'\}$, we let $\alpha(\theta)$ be the formula $\bigwedge_{i=j\in\theta} x_i = x_j$, for a conjunction η of equalities of elements in $\{\rho(1),\rho(1').\rho(2),\rho(2'),\rho(3),\rho(3')\}$, let $\beta(\eta)$ be the formula $\bigwedge_{\rho(i)=\rho(j)\in\eta} \sim(x_i,x_j)$, and for a conjunction θ^c of equalities between an object in O and an element in $\{1,1',2,2',3,3'\}$ we let $\alpha(\theta^c) = \bigwedge_{o=i\in\theta^c} o = x_i$.

In order to construct formula φ_e, we distinguish 2 types of joins:

- Joins of form $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$ where all of i,j,k belong to either $\{1,2,3\}$ or $\{1',2',3'\}$.

  Assume that i,j,k belong to $\{1,2,3\}$ (the other case is of course symmetrical). We first consider the case in which $\theta_{\bowtie}$ has no equalities, while $\eta_{\bowtie}$ has three equalities. Moreover, assume for the sake of readability that $\eta_{\bowtie} = (\rho(1) = \rho(1')) \wedge (\rho(2) = \rho(2')) \wedge (\rho(3) = \rho(3'))$. We then let

$$\begin{aligned}\varphi_e(x_i,x_j,x_k) = \varphi_{e_1}(x_1,x_2,x_3) \wedge \alpha(\theta_\ell) \wedge \alpha(\theta^c_\ell) \wedge \beta(\eta_\ell) \wedge \\ \exists w \Big(\sim(x_1,w) \wedge \exists x_1 \big(\sim(x_2,x_1) \ \wedge \exists x_2 (\sim(x_3,x_2) \varphi_{e_2}(w,x_1,x_2) \wedge \\ \alpha(\theta_r)[x_{1'},x_{2'},x_{3'} \to w,x_1,x_2] \wedge \alpha(\theta^c_r)[x_{1'},x_{2'},x_{3'} \to w,x_1,x_2] \wedge \\ \beta(\eta_r)[x_{1'},x_{2'},x_{3'} \to w,x_1,x_2]) \big) \Big)\end{aligned}$$

  Where a formula $\psi[x,y,z \to x',y',z']$ is just the formula ψ in which we replace each occurrence of variables x,y,z for x',y',z', respectively. For the case when $\theta_{\bowtie}$ is nonempty, notice here than any equality in $\theta_{\bowtie}$ only makes our life easier, since it eliminates one of the existential guesses we need in the above formula. Furthermore, if $\eta_{\bowtie}$ has less equalities, then we just remove the corresponding $\sim$ predicates. This cover all other possible cases of $\theta_{\bowtie}$ and $\eta_{\bowtie}$.

  Let us illustrate this construction with an example.

Consider the expression $e = e_1 \bowtie^{1,2,3}_{1=2\wedge\rho(2)=\rho(2')\wedge\rho(2')=\rho(3')} e_2$. Then θ_ℓ is $1=2$, $\eta_\bowtie$ is $\rho(2)=\rho(2')$ and $\eta_r = \rho(2') = \rho(3')$, all of the remaining formulas being empty. Then we have:

$$\varphi_e(x_1,x_2,x_3) = \varphi_{e_1}(x_1,x_2,x_3) \wedge x_1 = x_2 \wedge \exists w \Big(\exists x_1 \big(\sim(x_1,x_2) \wedge \exists x_2 (\varphi_{e_2}(w,x_1,x_2) \wedge \sim(x_1,x_2)) \big) \Big)$$

- Joins of form $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$ where not all of i,j,k belong to either $\{1,2,3\}$ or $\{1',2',3'\}$. Assume for the sake of readability that $i=1$, $j=2$ and $k=3'$ (all of other cases are completely symmetrical). We have again two possibilities.

  (-) There are no equalities in $\theta_\bowtie$. Assume that $\eta_\bowtie = (\rho(1)=\rho(1')) \wedge (\rho(2)=\rho(2')) \wedge (\rho(3)=\rho(3'))$ (we have already proved that there are at most 3 equalities in η'), cases with less equalities are treated along the same lines. We then let

$$\varphi_e(x_1,x_2,x_{3'}) =$$
$$\big(\exists x_3(\varphi_{e_1}(x_1,x_2,x_3) \wedge \alpha(\theta_\ell) \wedge \alpha(\theta^c_\ell) \wedge \beta(\eta_\ell)) \wedge \sim(x_3,x_{3'})\big) \wedge \exists x_3 \Big(\sim(x_1,x_3) \wedge \exists x_1 \big(\sim(x_2,x_1) \wedge \varphi_{e_2}(x_3,x_1,x_{3'}) \wedge \alpha(\theta_r)[x_{1'},x_{2'} \to x_3,x_1] \wedge \alpha(\theta^c_r)[x_{1'},x_{2'} \to x_3,x_1] \wedge \beta(\eta_r)[x_{1'},x_{2'} \to x_3,x_1] \big) \Big)$$

  (-) There is a single equality in $\theta_\bowtie$. Assume for the sake of readability that $i=1$, $j=2$ and $k=3'$ (all of other cases are completely symmetrical). Notice that if $\theta_\bowtie$ has the equality $3=3'$, then this is equivalent to the previous case with one equality in $\theta_\bowtie$, but with $k=3$. Moreover, equalities in $\theta_\bowtie$ involving 1 or 2 just make our life easier, so we will also not take them into account here. We are thus left with the assumption that $\theta_\bowtie$ contains the equality $3=1'$ (the case where it contains instead $3=2'$ is symmetrical)

  Moreover, assume as well that $\eta_\bowtie = (\rho(1)=\rho(1')) \wedge (\rho(2)=\rho(2')) \wedge (\rho(3)=\rho(3'))$ (we have already proved that there are at most 3 equalities in $\eta_\bowtie$, and from the form of the formula it is clear that all other cases are treated along the same lines).

  We then let

$$\varphi_e(x_1,x_2,x_{3'}) =$$
$$\exists x_{1'} \Big(\varphi_{e_1}(x_1,x_2,x_{1'}) \wedge \alpha(\theta_\ell)[x_3 \to x_{1'}] \wedge \alpha(\theta^c_\ell)[x_3 \to x_{1'}] \wedge \beta(\eta_\ell)[x_3 \to x_{1'}] \wedge \sim(x_1,x_{1'}) \wedge \exists x_1 \big(\sim(x_1,x_2) \wedge \varphi_{e_2}(x_{1'},x_1,x_{3'}) \wedge x_{1'} = x_{3'} \wedge \alpha(\theta_r)[x_{2'} \to x_1] \wedge \alpha(\theta^c_r)[x_{2'} \to x_1] \wedge \beta(\eta_r)[x_{2'} \to x_1] \big) \Big)$$

Having established how to construct φ_e, it is now straightforward to show that it satisfies the property of Lemma 8.5.2. It is also readily observed that the size of formula φ_e corresponding to e is $O(|e|)$. □

To finish the proof of Proposition 8.5.1 we show that FO^4 formulas can be evaluated efficiently.

Lemma 8.5.3. *Let φ be an arbitrary formula using at most four variables. Then the set of all tuples that make φ true in $\mathcal{M}$, with $\mathcal{M}$ as above (we omit the subscript T for the sake of readability, since it is now clear), can be computed in time $O(|F| \cdot |O|^4)$.*

Proof. To see that this holds note that we can assume that our formulas only use the connectives $\neg, \vee$ and the quantifier $\exists$. Indeed, we can assume this since any formula using other quantifiers can be rewritten using the ones above with a constant blow-up in the size of formula. In particular, our formulas in Lemma 8.5.2 use only $\wedge$ in addition to these three logical connectives, and $\wedge$ can be rewritten in terms of $\vee$ and $\neg$.

The desired algorithm works as follows.

1. Build a parse tree for the formula φ.
2. Compute the output relation(s) bottom-up using the tree.

To see that the algorithm works with the desired time bound we only have to make sure that each of the computation steps in 2 can be performed in time $O(|O|^4)$. We have three cases to consider, based on whether we are using negation, disjunction, or existential quantification. Here we assume that we compute a matrix $\psi(\mathcal{M})$, for each subformula ψ of φ. Note that, since we use formulas with at most four free variables each matrix can be of size at most $|O|^4$ (i.e. we are working with a four dimensional matrix). If the (sub)formula has only two free variables the resulting matrix will, of course, be two dimensional.

First we consider the case of negation. That is, assume that we have a matrix $\psi(\mathcal{M})$ and we are evaluating a formula $\varphi = \neg\psi$. Then we simply build a matrix for the $\varphi(\mathcal{M})$ by flipping each bit in the matrix for $\psi(\mathcal{M})$. This can clearly be done in time $O(|O|^4)$ by traversing the entire matrix.

Next, consider the case when $\varphi = \exists x\psi(x,y,z,w)$ and assume that we have the matrix for $\psi(x,y,z,w)$. The existing matrix is now reduced to a three dimensional matrix with the value 1 in position i,j,k if and only if there is an l such that $\psi(\mathcal{M})[l,i,j,k] = 1$. Note that computing this amounts to scanning the entire matrix for ψ. In the case when ψ case only three free variables we will need only $O(|O|^3)$ time to compute $\varphi(\mathcal{M})$.

Finally, let $\varphi = \psi_1(x,y,w) \vee \psi_2(x,y,z,w)$. The cases when ψ_1 and ψ_2 have a different number of free variables follows by symmetry. What we do first is to compute a 4-D matrix

$\psi'_1(\mathcal{M})$ by setting $\psi'_1(\mathcal{M})[i,j,k,l] = 1$ iff $\psi_1(\mathcal{M})[i,j,l] = 1$. Note that this matrix can be computed in time $O(|O|^4)$. Next we compute the output matrix by putting 1 in each cell where either $\psi'_1(\mathcal{M})$ or $\psi_2(\mathcal{M})$ have 1. All the other cases can be performed symmetrically by using the appropriate matrices and their projections.

This completes the proof of Lemma 8.5.3. □

Navigational fragments To pose navigational queries, one needs the recursive algebra, so the question is whether similar bounds can be obtained for meaningful fragments of TriAL*. Using the ideas from the proof of Theorem 8.4.2 we immediately get an $O(|e| \cdot |O| \cdot |T|^2)$ upper bound for TriAL$^=$ with recursion. However, we can improve this result for the fragment reachTA$^=$ that extends TriAL$^=$ with essentially *reachability* properties, such as those used in RPQs and similar query languages for graph databases.

To define it, we restrict the star operator to mimic the following graph database reachability queries:

- the query "reachable by an arbitrary path", expressed by $(R \bowtie^{1,2,3'}_{3=1'})^*$; and
- the query "reachable by a path labeled with the same element", expressed by $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$.

These are the only applications of the Kleene star permitted in reachTA$^=$. For this fragment, we have the same lower complexity bound.

Proposition 8.5.4. *The problem* QUERYCOMPUTATION *for* reachTA$^=$ *can be solved in time* $O(|e| \cdot |O| \cdot |T|)$.

Proof. To show this we will use the algorithm presented in Proposition 8.5.1. All of the operations except the evaluation of Kleene star will be preformed in a same way as there. Note that we can assume this since the algorithm in Lemma 8.5.3 computes the subexpressions bottom up using the matrices representing the output. Thus we can use it to compute answers to subformulas, compose it with the method presented here to evaluate Kleene stars and proceed with the algorithm from Lemma 8.5.3. To obtain the desired complexity bound we only have to show how to compute navigational operations in time $O(|O| \cdot |T|)$.

That is, we show how to, given a matrix representation for a relation R we compute matrix representation for $(R \bowtie^{1,2,3'}_{3=1'})^*$ and $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$, respectively.

Let $O = \{o_1, \ldots, o_n\}$ be the set of object appearing in our triplestore T. (The assumption that they are ordered is standard when considering matrix representations). As input, we are given a three dimensional matrix R representing the output of relation R when evaluated over T. That is we have $(o_i, o_j, o_k) \in R(T)$ if and only if $R[i,j,k] = 1$. (Here we use R both to denote the relation R and its matrix representation).

First we give a procedure that computes the matrix M_e for the expression

$$e = (R \bowtie^{1,2,3'}_{3=1'})^*.$$

Procedure 3 Computing $e = (R \bowtie^{1,2,3'}_{3=1'})^*$

Input: Matrix representation of R

Output: Matrix M_e representing e

1: Precomputing the reachability matrix R_{reach}:
2: **for** $i = 1 \to n$ **do**
3: **for** $j = 1 \to n$ **do**
4: **for** $k = 1 \to n$ **do**
5: **if** $R[i,k,j] = 1$ **then**
6: $R_{reach}[i,j] = 1$
7: Compute the transitive closure R^*_{reach}
8: Compute the output matrix M_e:
9: **for** $i = 1 \to n$ **do**
10: **for** $j = 1 \to n$ **do**
11: **for** $k = 1 \to n$ **do**
12: **if** $R[i,k,j] = 1$ **then**
13: **for** $l = 1 \to n$ **do**
14: **if** $R^*_{reach}[j,l] = 1$ **then**
15: $M_e[i,k,l] = 1$

To show that the algorithm works correctly notice that steps 1 to 6 precompute the matrix R_{reach} such that $R_{reach}[i,j] = 1$ if and only if o_i has and out edge ending in o_j (or equivalently $(o_i, o, o_k) \in T$ for some o). After this in step 7 we compute the transitive closure R^*_{reach} thus obtaining all pairs of nodes reachable one from another using path of arbitrary label in the graph representing T. Next in steps 8 to 15 we simply compute all the triples in the output matrix M_e. To do so we observe that a pair (o_i, o_k) will belong to some triple (o_i, o_k, o_l) of the output, if there is j such that $(o_i, o_k, o_j) \in T$ (line 12) and o_l is reachable from o_j (line 14).

To determine the complexity of the algorithm notice that steps 1 to 6 take time $O(|O|^3) = O(|T|)$, while computing the transitive closure in step 7 can be done using Warshall's algorithm (see T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, 2003.) in time $O(|O|^3) = O(|T|)$. Finally steps 8 to 15 take time $O(|O| \cdot |T|)$, thus giving us the desired time bound.

Next we show how to compute joins of the form $(R \bowtie^{1,2,3'}_{3=1',2=2'})^*$ using a slight modification of the algorithm above.

Procedure 4 Computing $e = (R \bowtie^{1,2,3'}_{3=1',2=2'})^*$

Input: Matrix representation of R

Output: Matrix M_e representing e

```
1:  for k = 1 → n do
2:      Precomputing the reachability matrix R^k_reach:
3:      for i = 1 → n do
4:          for j = 1 → n do
5:              if R[i,k,j] = 1 then
6:                  R_reach[i,j] = 1
7:      Compute the transitive closure (R^k_reach)*
8:      compute the output matrix M_e:
9:      for i = 1 → n do
10:         for j = 1 → n do
11:             if R[i,k,j] = 1 then
12:                 for l = 1 → n do
13:                     if (R^k_reach)*[j,l] = 1 then
14:                         M_e[i,k,l] = 1
```

It is straightforward to see that the algorithm uses the same time to compute the output as the algorithm in Procedure 3.

To show that it works correctly observe that we precompute matrix R^k_{reach} for each k, thus checking reachability only for triples whose second node is o_k. Since the rest of the algorithm works in the same way as the one in Procedure 3, we conclude that the computed answer M_e represents e correctly. □

8.6 Expressive power

In this section we compare the expressive power of TriAL with that of classical relational languages. As already mentioned, FO is one of the most common database yardsticks when it comes to relational querying and close connections with it are often one of the priorities in query language design.

Here we will show that power of TriAL and its recursive variant TriAL* is precisely bounded by well studied fragments of FO and transitive closure logic TrCl [Grädel, 1991, Libkin, 2004]. In particular, we show that TriAL lives between FO^3 and FO^6, while being uncomparable to FO^4 and FO^5, the inclusions here being strict. The intuitive reason for this is that while triple joins can be simulated using six variables, at the same time they carrying more information in their conditions than fits into five variables. An analogous result holds for TriAL*, but this time

with TrCl^3 through TrCl^6. We will also show that the fragment that allows no inequalities, that is $\mathsf{TriAL}^=$, lies strictly between FO^3 and FO^4.

As usual, we say that a language $\mathcal{L}_1$ is contained in a language $\mathcal{L}_2$ if for every query in $\mathcal{L}_1$ there is an equivalent query in $\mathcal{L}_2$. If in addition $\mathcal{L}_2$ has a query not expressible in $\mathcal{L}_1$, then $\mathcal{L}_1$ is strictly contained in $\mathcal{L}_2$. The languages are equivalent if each is contained in the other. They are incomparable if none is contained in the other.

To compare TriAL with relational languages, we use exactly the same relational representation of triplestores as we did when we found Datalog fragments capturing TriAL and TriAL^*. That is, we compare the expressive power of TriAL with that of First–Order Logic (FO) over vocabulary $\langle E_1, \ldots, E_n, \sim \rangle$.

Since TriAL is a restriction of relational algebra, of course it is contained in FO. We do a more detailed analysis based on the number of variables. Recall that FO^k stands for FO restricted to k variables only. To give an intuition why such restrictions are relevant for us, consider, for instance, the join operation $e = E \bowtie^{1,3',3}_{2=2'} E$. It can be expressed by the following FO^6 formula: $\varphi(x_1, x_{3'}, x_3) = \exists x_2 \exists x_{1'} \exists x_{2'} \big(E(x_1, x_2, x_3) \wedge E(x_{1'}, x_{2'}, x_{3'}) \wedge x_2 = x_{2'}\big)$. This suggests that we can simulate joins using only six variables, and this extends rather easily to the whole algebra. One can furthermore show that the containment is proper in this case.

What about fragments of FO using fewer variables? Clearly we cannot go below three variables. It is not difficult to show that TriAL simulates FO^3, but the relationship with the 4 and 5 variable formalisms appears much more intricate, and its study requires more involved techniques. We can show the following.

Theorem 8.6.1.

- TriAL *is strictly contained in* FO^6.
- FO^3 *is strictly contained in* TriAL.
- TriAL *is incomparable with* FO^4 *and* FO^5.

The containment of FO^3 in TriAL is proved by induction, and we use pebble games to show that such containment is proper. For the last, more involved part of the theorem, we first show that TriAL is not contained in FO^5. Notice that the expression e given by

$$U \overset{1,2,3}{\underset{\theta}{\bowtie}} U, \text{ with } \theta = \{i \neq j \mid i, j \in \{1, 1', 2, 2', 3, 3'\}, i \neq j\},$$

is such that $e(T)$ is not empty if and only if T has six different objects (recall that U is the set of all triples (o_1, o_2, o_3) so that each o_i occurs in a triple in T). It then follows that TriAL is not contained in FO^5 (nor FO^4), cf. [Libkin, 2004]. To show that FO^4 is not contained in TriAL, we devise a game that characterizes expressibility of TriAL, and use this game to show that TriAL cannot express the following FO^4 query $\varphi(x, y, z)$:

$$\exists w \big(\psi(x, y, w) \wedge \psi(x, w, z) \wedge \psi(w, y, z) \wedge \psi(x, y, z)\big),$$

where

$$\psi(x,y,z) = \exists w\big(E(x,w,y) \wedge E(y,w,z) \wedge E(z,w,x)\big).$$

The above result also shows that TriAL cannot express all conjunctive queries, since in particular the query $\varphi(x,y,z)$ is a conjunctive query. This is of course expected; the intuition is that TriAL queries have limited memory and thus cannot express queries such as the existence of a k-clique, for large values of k.

Next we give the full proof.

Proof of Theorem 8.6.1. We split the proof into three parts, each corresponding to one of the claims of the theorem.

Part 1 Let e be a TriAL expression. We construct an FO^6 formula φ_e such that $e(T) = \varphi_e(I_T)$, for each triplestore T. The proof is by induction.

- For the base case, if e corresponds to a triplestore name E, then φ_e is $E(x,y,z)$.
- If $e = e_1 \cup e_2$, then $\varphi_e(x,y,z) = \varphi_{e_1}(x,y,z) \vee \varphi_{e_2}(x,y,z)$, which clearly is in FO^6 since existential variables within φ_{e_1} and φ_{e_2} can be renamed and reused.
- If $e = e_1 - e_2$, then $\varphi_e(x,y,z) = \varphi_{e_1}(x,y,z) \wedge \neg\varphi_{e_2}(x,y,z)$
- If $e = e_1 \bowtie^{i,j,k}_{\theta,\eta} e_2$, then $\varphi_e(x_i,x_j,x_k) = \exists x_u \exists x_v \exists x_w \varphi_{e_1}(x_1,x_2,x_3) \wedge \varphi_{e_2}(x_{1'},x_{2'},x_{3'}) \wedge \alpha(\theta) \wedge \beta(\eta)$, where u,v,w are the remaining elements that together with i,j,k complete $\{1,1',2,2',3,3'\}$, $\alpha(\theta)$ contains the equality $x_p = x_q$ or $x_p = o$ for each equality $p = q$ or $p = o$ in θ, for $o \in O$ and $p,q \in \{1,1',2,2',3,3'\}$, and likewise for inequalities, and $\beta(\eta)$ contains atom $\sim(x_p,x_q)$ for each equality $\rho(p) = \rho(q)$ in η, and likewise for inequalities using atom $\neg\sim$.
- Similarly, if $e = \sigma_{\theta,\eta} e_1$ then $\varphi_e(x,y,z) = \varphi_{e_1}(x,y,z) \wedge \alpha(\theta) \wedge \beta(\eta)$, where $\alpha(\theta)$ and $\beta(\eta)$ are defined as in the previous bullet.

It is now straightforward to check the desired properties for e and φ_e.

That the containment is strict follows from Part 3 of the proof.

Part 2 To show that FO^3 is contained in TriAL, one needs to show how to construct, for every FO^3 formula φ, an equivalent TriAL expression e_φ such that $e_\varphi(T) = \varphi(I_T)$, for all triplestores T.

The construction is done by induction on the formula.

Recall here that U is just a shorthand for the relation that contains O^3.

- For the base case, if $\varphi = E(x_1,x_2,x_3)$ for some triplestore name then e_φ is just E. However, in the general case when $\varphi = E(x_i,x_j,x_k)$, for each of i,j,k in $\{1,2,3\}$, we let $e_\varphi = E \bowtie^{i,j,k} E$. For the other base case, if φ is $x_1 = x_2$, then $e_\varphi = \sigma_{1=2} U$.

- If $\varphi = \neg\varphi_1$, then $e_\varphi = U - e_{\varphi_1}$ (recall that we assume active domain semantics for FO formula).
- If $\varphi = \exists x\varphi_1(\bar{y})$, then $e_\varphi = e_{\varphi_1} \bowtie^{\bar{d}} U$, where $\bar{d}$ depends on the size of $\bar{y}$: if $|\bar{y}| = 3$ then $\bar{d} = i, j, k'$, if $\bar{y} = 2$ then $\bar{d} = i, j', k'$, and if $\bar{y} = 1$ then $\bar{d} = i', j', k'$.
- If $\varphi = \varphi_1(\bar{x},\bar{y}) \vee \varphi_2(\bar{x},\bar{z})$, then $e_\varphi = e_{\varphi_1} \cup e_{\varphi_2}$. Notice here that we assume that variables in $\bar{x},\bar{y},\bar{z}$ appear in the same order in both φ_1 and φ_2. If this is not the case then one can only permute the variables by doing a join, as in the base case.

We leave the proof that φ and e_φ satisfy our desired properties, since it is easy to check. The key idea is that we do not need projection in our algebra to simulate FO^3 queries, since we know that they will have 3 free variables at the end, in the induction step we can just ignore some of the positions in the triples.

To show that the containment is proper, consider the following property over triplestore databases:

A triplestore database T has four different objects.

It is not difficult to construct a TriAL expression e such that $e(T)$ is nonempty if and only if T has four different objects. For example, one can use the expression $e = U \bowtie^{1,2,3}_{\theta} U$, where $\theta = (1 \neq 2) \wedge (1 \neq 3) \wedge (1 \neq 1') \wedge (2 \neq 3) \wedge (2 \neq 1') \wedge (3 \neq 1')$.

On the other hand, let $T_3 = (O_3, E_3, \rho)$ be the triplestore in which $O_3 = \{a, b, c\}$ and $E_3 = O_3 \times O_3 \times O_3$, and $T_4 = (O_4, E_4, \rho')$ be the triplestore in which $O_4 = \{a, b, c, d\}$ and $E_4 = O_4 \times O_4 \times O_4$. In addition we set $\rho(a) = \rho(b) = \rho(c) = 1$ and $\rho' = \rho \cup \{(d, 1)\}$. It is trivial to show that these structures cannot be distinguished by any formula in the infinitary logic $\mathcal{L}^3_{\infty\omega}$ [Libkin, 2004], since the duplicator always has a strategy to ensure that the 3-pebble game can be played forever in these structures (see e.g. [Libkin, 2004]). Note that the standard game will work here, since all the data values are the same, so they do not influence the winning strategy of the duplicator. It follows that the expression e cannot be expressed in FO^3 (in fact, not even in $\mathcal{L}^3_{\infty\omega}$).

Part 3 For Part 3, we show that TriAL is incomparable with FO^4 and FO^5.

We begin by showing that the following TriAL query:

$$e_6 := U \overset{1,2,3}{\underset{\theta}{\bowtie}} U, \text{ with } \theta = \bigwedge_{i,j \in \{1,2,3,1',2',3'\}, i \neq j} i \neq j,$$

cannot be expressed in FO^5 (and thus not in FO^4).

Note that this is a modification of the query from part 1 of this proof that simply states that our triplestore has at least six objects. Now take $T_5 = (O_5, E_5, \rho)$ with $O_5 = \{a, b, c, d, e\}$,

and $E_5 = O_5 \times O_5 \times O_5$, where ρ assigns the same data value to all elements of O_5 and define O_6 in an analogous way, but with six elements. It is a well known fact [Libkin, 2004] that the duplicator has a winning strategy in a 5-pebble game on these two structures, so they can not be distinguished by an FO^5 formula. On the other hand our expression e_6 does distinguish them and is thus not expressible in FO^5.

Next we show that there is an FO^4 expression that cannot be expressed by any TriAL query (and thus TriAL cannot express neither full FO^5 nor FO^6). In order to do that, we first need to show that triple algebra expressions can be expressed with a particular extension of FO^3, that we call here FO^3-join.

Formally, we construct FO^3-join formulas from FO^3 formulas, the usual operators of disjunction, conjunction, negation, existential and universal quantification, and the following join operator: if φ_1 and φ_2 are formulas in FO^3-join that use variables x_1, x_2, x_3 and $x_{1'}, x_{2'}, x_{3'}$ respectively, θ is a conjunction of equalities between indexes in $\{1, 1', 2, 2', 3, 3'\}$ and η is a conjunction of equalities between indexes in $\rho(1), \ldots, \rho(3')$, then the formula $\varphi(x_i, x_j, x_k) = \varphi_1(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(x_{1'}, x_{2'}, x_{3'})$ is a formula in FO^3-join that only uses variables x_i, x_j, x_k. Furthermore, the number of variables in FO^3-join formulas is restricted to 3, but note that for the sake of counting variables the construct $\varphi(x_i, x_j, x_k) = \varphi_1(x_1, x_2, x_3) \bowtie^{i,j,k}_{\theta,\eta} \varphi_2(x_{1'}, x_{2'}, x_{3'})$ is assumed to use only variables x_i, x_j and x_k.

The semantics of the join construct is defined in the same way as for Triple Algebra, and the rest of the operators are defined in the same way as in FO. It is now not difficult to show the following:

Lemma 8.6.2. *Triple Algebra is contained in* FO^3*-join.*

In fact, one can actually show that both languages have the same expressive power, but for the sake of this proof we will not bother. Continuing with the proof, we now define a game that characterizes expressibility in FO^3-join.

Let J be the set of all the join symbols that we allow in TriAL. A *recipe* p for FO^3-join is a tree of rank 2 (i.e., every node can have at most two children) labeled with symbols from alphabet $\{\exists, \forall\} \cup \mathcal{J}$, such that the following holds: If a node n of p has two children, then it is labeled with a symbol in J, and if a node n of p has one child, then it is labeled with $\exists$ or $\forall$.

For every such recipe p, define the *quantifier class* $L(p)$ inductively as follows:

- $L(\varepsilon)$ contains quantifier and join free formulae.
- If the root of p is labeled with $Q \in \{\exists, \forall\}$, then $L(p)$ is the closure under conjunctions and disjunctions of the class $L(p') \cup \{Qx\varphi \mid \varphi \in L(p')\}$, where p' is the subtree of p whose root is the only child of p.

- If the root of p is labeled with a symbol $\bowtie$ in J, let p_1 and p_2 be the subtrees of p whose roots are the first and the second child of p, respectively. Then $L(p)$ is the closure under conjunctions and disjunctions of the class of all formulae $\varphi \bowtie \psi$, where $\varphi \in L(p_1)$ and $\psi \in L(p_2)$.

We now define the join game between two structures. This game proceeds as in a typical 3-pebble game (see [Libkin, 2004] for a precise explanation), except the following sets of moves are available to the spoiler:

The join $\bowtie^{i,j,k}_{\theta,\eta}$ move:
The spoiler picks a structure, and then splits the 3 pebbles in that structure into two sets of 3 pebbles, set 1 and set 2, with the condition that the split *satisfies* the join: If before the move the first, second and third pebbles where in elements a, b and c, then the first, second and third elements of each of the set of pebbles must be placed in elements a_1, b_1, c_1 and a_2, b_2, c_2 such that $(a,b,c) = (a_1,b_1,c_1) \bowtie^{i,j,k}_{\theta,\eta} (a_2,b_2,c_2)$.

Duplicator must then split the pebbles in the other structure into two sets of pebbles, in the same fashion as the spoiler, with the split also satisfying the conditions of the join, Spoiler then picks either set 1 or set 2, and remove the other set of pebbles from both structures.

A *join game* on a pair of structures $(\mathcal{A},\mathcal{B})$, is played as the regular 3 pebble game, except now the spoiler can use any number of $\bowtie$ moves, for $\bowtie$ in J. The winning conditions for both players are the same as in the 3-pebble game. For every recipe p of FO^3-join we also define the $L(p)$-join game. This contains all join games in which the sequence of moves performed by the spoiler are described by a path from the root of p to one of its leaves.

Let L be a class of FO^3-join formulae and A and B structures of vocabulary $\langle E, \sim \rangle$. We write $\mathcal{A} \preceq_L \mathcal{B}$ if $\mathcal{A} \models \varphi$ implies $\mathcal{B} \models \varphi$, for every sentence $\varphi \in L$.

Lemma 8.6.3. *The following are equivalent:*

- *The duplicator has a winning strategy on all $L(p)$ join games.*
- $\mathcal{A} \preceq_{L(p)} \mathcal{B}$

Before we prove this Lemma, we make the following crucial observation: If, in a join game a pebble has already been placed on element $a \in \mathcal{A}$, then the remainder of the game can be considered as a game with two pebbles on $(\mathcal{A}, a)$, until the first pebble is replaced somewhere else, or a join move are performed. We call these games *truncated*.

Proof. We prove the contrary: If there is a sentence φ of class $L(p)$ such that $\mathcal{A} \models \varphi$ but $\mathcal{B} \not\models \varphi$, then the spoiler has a winning strategy for the $L(p)$-join game.

We prove this by induction on the height of p.

The case when p is empty is trivial.

Assume that Lemma holds for all recipes of height k, and let p be a recipe of height $k+1$. Furthermore, assume that there is a sentence φ such that $\mathcal{A} \models \varphi$, but $\mathcal{B} \not\models \varphi$. We will construct a winning strategy for the spoiler. If φ is a boolean combinations of formulas, then the two structures are distinguished by at least one of them. We are thus left with the following cases:

- φ is of form $\exists\psi(\bar{x})$, where $\bar{x}$ is a tuple of at most two variables, and ψ has depth at most $k-1$ and belongs to $L(q)$, where q is the subtree whose root is the single child of p. Then the spoiler can win as follows. In his first move he places one pebble in element a such that $(\mathcal{A}, a) \models \psi$. No matter in which element $b \in \mathcal{B}$ the duplicator places its pebble, we know that $(\mathcal{B}, b) \not\models \psi$, and thus the spoiler has a winning strategy for the remainder of the truncated game.

- φ is of form $\forall\psi(\bar{x})$, in which case the strategy is analogous to the previous one

- $\varphi(a,b,c)$ is of form $\varphi_1 \bowtie \varphi_2$, for some $\bowtie$ in J (note that a,b,c are interpreted as constants of A and B). Then p has two children p_1 and p_2, both of height $\leq k$, and $\varphi_1 \in L(p_1)$, $\varphi_2 \in L(p_2)$. Since $\mathcal{A} \models \varphi(a,b,c)$, yet $\mathcal{B} \not\models \varphi(a,b,c)$, spoiler can win by first placing pebbles on elements a,b,c, and splitting pebbles placing them into sets (a_1,b_1,c_1) and (a_2,b_2,c_2) of elements in A such that $(a_1,b_1,c_1) \bowtie (a_2,b_2,c_2) = (a,b,c)$. Given that $\mathcal{B} \not\models \varphi(a,b,c)$, then for every pair (d_1,e_1,f_1) and (d_2,e_2,f_2) of elements in B such that $(d_1,e_1,f_1) \bowtie (d_2,e_2,f_2) = (a,b,c)$, it must be the case that either $(\mathcal{B} \not\models \varphi_1(d_1,e_1,f_1)$ or $(\mathcal{B} \not\models \varphi(d_2,e_2,f_2)$. Depending on the move of the duplicator, spoiler chooses the set accordingly, and continues to win the truncated game on $(\mathcal{A}, a_i, b_i, c_i)$ and $(\mathcal{B}, d_i, e_i, f_i)$, for $i = 1$ or $i = 2$.

□

We now continue with the proof of the Theorem. Due to Lemma 8.6.3, all that is left to do is to show structures A and B such that the duplicator can win any join game, and yet they are distinguished by an FO^4 formula.

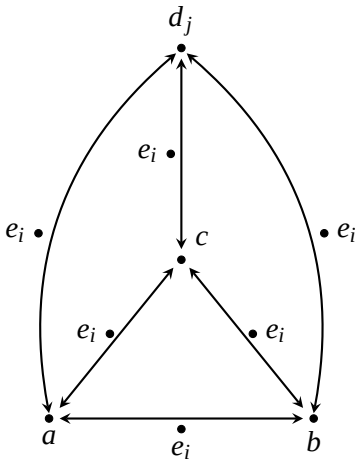
The structures are as follows:

Consider objects a,b,c plus objects $d_1,\ldots,d_9$ and $e_1,\ldots,e_{12}$.

- Structure A contain edges $(a,e_i,b),(b,e_i,a),(a,e_i,c),(c,e_i,a),(b,e_i,c),(c,e_i,b)$, for each $1 \leq i \leq 12$, plus edges $(a,e_i,d_j),(d_j,e_i,a),(b,e_i,d_j),(d_j,e_i,b),(c,e_i,d_j),(d_j,e_i,c)$ for each $1 \leq i \leq 4$ and $1 \leq j \leq 12$.

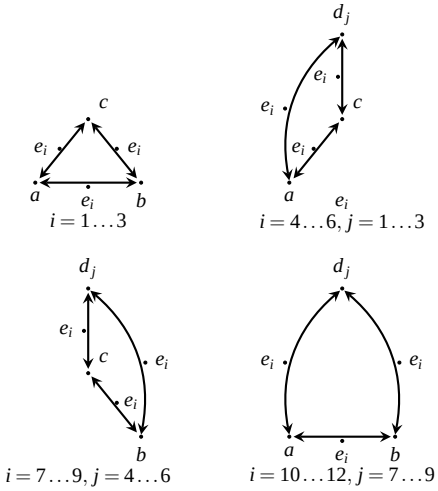
- Structure B also has edges $(a,e_i,b),(b,e_i,a),(a,e_i,c),(c,e_i,a),(b,e_i,c),(c,e_i,b)$, for each $1 \leq i \leq 3$, plus edges $(a,e_i,b),(b,e_i,a),(b,e_i,d_j),(d_j,e_i,b)$ and $(a,e_i,d_j),(d_j,e_i,a)$ for each $1 \leq j \leq 3$ and for each $4 \leq i \leq 6$; $(a,e_i,c),(c,e_i,a),(d_j,e_i,c),(c,e_i,d_j)$

and $(a,e_i,d_j),(d_j,e_i,a)$ for each $4 \leq j \leq 6$ and for each $7 \leq i \leq 9$; and $(b,e_i,c),(c,e_i,b),(b,e_i,d_j),(d_j,e_i,b)$ plus $(c,e_i,d_j),(d_j,e_i,c)$ for each $7 \leq j \leq 9$ and for each $10 \leq i \leq 12$.



$i = 1 \ldots 12, j = 1 \ldots 4$

Structure A



Structure B

It is not difficult to see that the duplicator has a winning strategy for the standard 3-pebble games on this structure. If the three pebbles placed by the spoiler do not correspond with an edge of the structure, the the duplicator just mimics the same moves, the partial isomorphism trivially holds. If the third pebble correspond to some edge of form (u,e_i,v), for u and v in $\{a,b,c,d_1,\ldots,d_9\}$ and $1 \leq i \leq 12$ in A that is not in B, assume the pebble was last placed in u (other two cases are symmetrical). Then the duplicator needs to find a permutation τ of the objects in A, such that $\tau(e_i) = e_i$, $\tau(v) = v$, $\tau(\mathcal{A})$ is isomorphic to $\mathcal{A}$ and the edge $(\tau(u),\tau(e_i),\tau(v))$ is in B, and place pebbles in $(\tau(u),\tau(e_i),\tau(v))$, so that the partial isomorphism still holds. For the remainder of the game, duplicator acts as if dealing with $\tau(\mathcal{A})$ instead of A.

Next, for the i,j,k-join move, assume that pebbles in structures $\mathcal{A}$ and $\mathcal{B}$ are in elements a_i,a_j,a_k and b_i,b_j,b_k, respectively. If spoiler divides first structure B duplicator just responds with the same edges in A. Now if spoiler divides structure A into pebbles (a_1,a_2,a_3) and $(a_{1'},a_{2'},a_{3'})$ satisfying the join condition, we have three cases:

- If none of (a_1,a_2,a_3) and $(a_{1'},a_{2'},a_{3'})$ are edges in A then duplicator mimics the pebble placement.
- If, say, only (a_1,a_2,a_3) is an edge in A, then the duplicator proceeds like in the above paragraph.
- Otherwise, if both (a_1,a_2,a_3) and $(a_{1'},a_{2'},a_{3'})$ are edges in A, duplicator needs to find a permutation τ of the objects in A such that $\tau(\mathcal{A})$ is isomorphic to A; $\tau(a_i) = a_i$, $\tau(a_j) = a_j$, and $\tau(a_k) = a_k$; and edges $(\tau(a_1),\tau(a_2),\tau(a_3))$ and $(\tau(a_4),\tau(a_5),\tau(a_6))$ belong to B, and respond with those pebbles. The partial isomorphisms trivially holds.

All that is left to show that this is a winning strategy for the duplicator is to show that there are always such permutations, no matter where are the pebbles placed. This can be easily shown with a lengthy and straightforward case by case analysis.

From Lemma 8.6.3 we obtain that A and B agree on all FO^3-join formulas. However, it is not difficult to see that they do not agree to the following FO^4 formula (which is only true in A):

$$\varphi(x,y,z) = \exists x \exists y \exists z \exists w \big(\psi(x,y,w) \wedge \psi(x,w,z) \wedge \psi(w,y,z) \wedge \psi(x,y,z) \wedge x \neq y \wedge x \neq z \wedge x \neq w \wedge y \neq z \wedge y \neq w \wedge z \neq w\big),$$

where

$$\psi(x,y,z) = \exists w \big(E(x,w,y) \wedge E(y,w,x) \wedge E(y,w,z) \wedge E(x,w,y) \wedge E(x,w,z) \wedge E(z,w,x) \wedge x \neq z \wedge x \neq y \wedge y \neq z\big).$$

This shows that FO^4 is not contained in TriAL, which completes the proof of Part 3. □

Expressivity of TriAL$^=$ The TriAL queries we used to separate it from FO^5 or FO^4 make use of inequalities in the join conditions. Thus, it is natural to ask what happens when we restrict our attention to TriAL$^=$, the fragment that disallows inequalities in selections and joins. We saw in Section 8.4 that this fragment appears to be more manageable in terms of query answering. This suggests that fewer variables may be enough, as the number of variables is often indicative of the complexity of query evaluation [Immerman and Kozen, 1989, Vardi, 1995]. This is indeed the case.

Theorem 8.6.4.

- FO^3 *is strictly contained in* TriAL$^=$.
- TriAL$^=$ *is strictly contained in* FO^4.

Proof. The containment of TriAL$^=$ in FO^4 was shown in the proof of Proposition 8.5.1, and that TriAL$^=$ contains FO^3 was already showed in the second part of the proof of Theorem 8.6.1, since the translation used there does not make use of inequalities in joins.

That the containments are strict follows from the proof of Theorem 8.6.1. □

Expressivity of the recursive algebra Next, we turn to the expressive power of TriAL*. Since the Kleene star essentially defines the transitive closure of join operators, it seems natural for our study to compare TriAL* with Transitive Closure Logic, or TrCl.

Formally, TrCl is defined as an extension of FO with the following operator. If $\varphi(\bar{x},\bar{y},\bar{z})$ is a formula, where $|\bar{x}| = |\bar{y}| = n$, and $\bar{u},\bar{v}$ are tuples of variables of the same length n, then $[\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x},\bar{y},\bar{z})](\bar{u},\bar{v})$ is a formula whose free variables are those in $\bar{z}$, $\bar{u}$ and $\bar{v}$. The semantics

is as follows. For an instance I and an assignment $\bar{c}$ for variables $\bar{z}$, construct a graph G whose nodes are elements of I^n and edges contain pairs $(\bar{u_1},\bar{u_2})$ so that $\varphi(\bar{u_1},\bar{u_2},\bar{c})$ holds in I. Then $I \models [\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x},\bar{y},\bar{c})](\bar{a},\bar{b})$ iff $(\bar{a},\bar{b})$ is in the transitive closure of this graph G.

It is fairly easy to show that TriAL* is contained in TrCl; the question is whether one can find analogs of Theorem 8.6.1 for fragments of TrCl using a limited number of variables. We denote by TrCl^k the restriction of TrCl to k variables. Note that constructs of form $[\mathbf{trcl}_{\bar{x},\bar{y}}\varphi(\bar{x},\bar{y},\bar{z})](\bar{t_1},\bar{t_2})$ can be defined using $|\bar{t_1}|+|\bar{t_2}|+|\bar{z}|$ variables, by reusing $\bar{t_1}$ and $\bar{t_2}$ in φ.

Then we can show that the relationship between TriAL* and TrCl mimics the results of Theorem 8.6.1 for the case of TriAL and FO.

Theorem 8.6.5.

- TriAL* *is strictly contained in* TrCl^6.
- TrCl^3 *is strictly contained in* TriAL*.
- TriAL* *is incomparable with* TrCl^4 *and* TrCl^5.

Proof. We split the proof into three parts, one for each of the claims.

Part 1 We begin by proving that TriAL* is strictly contained in TrCl^6. To see that TriAL* is contained in TrCl^6 we use induction on the structure of TriAL* expressions. Note that all the cases, except for the Kleene closure of various joins we use, are precisely the same translation as in the proof of Theorem 8.6.1. What remains to prove is that expressions of the form

$$e' := (e \underset{\theta,\eta}{\overset{i,j,k}{\bowtie}})^*$$

can be translated into TrCl^6 expressions (the other join being completely symmetrical).

To see this, let $\psi_e(x,y,z)$ be a TrCl^6 formula equivalent to e. That is we have that $I_T \models \psi_e(a,b,c)$ if and only if $(a,b,c) \in R(T)$, for any triplestore T, with I_T the FO-structure representing T. We define the following formula $\psi_{e'}(x',y',z')$ in TrCl^6:

$$\psi_e(x',y',z') \vee \exists x,y,z\ \big(\psi_e(x,y,z) \wedge [\mathbf{trcl}_{x,y,z,x',y',z'}\varphi(x,y,z,x',y',z')](x,y,z,x',y',z')\big)$$

Where $\varphi(x,y,z,x',y',z')$ is a formula such that $\varphi(a,b,c,a',b',c')$ holds in I_T iff there exists a triple (a'',b'',c'') such that $\psi_e(a'',b'',c'')$ holds and the join of (a,b,c) and (a'',b'',c'') produces triple (a',b',c'). The definition of this formula in TrCl^6 is rather cumbersome, since it depends on the positions i,j,k of the join in question. We just give two examples, the rest are treated in the same way: For the expression $e' = (e \bowtie^{1,2,3'})^*$, we have that $\varphi(x,y,z,x',y',z')$ is $x = x' \wedge y = y' \wedge \exists x' \exists y' \big(\psi_e(x,y,z) \wedge \psi_e(x',y',z')\big)$. As another example, if $e' = (e \bowtie^{1',2',3'})^*$, then φ is just $\psi_e(x,y,z) \wedge \psi_e(x',y',z')$.

Next we prove that $\psi_{e'}$ is equivalent to expression e' over all triplestores. For one direction, let T be a triplestore database using a set O of objects, and assume that triple (a,b,c) belong to $e'(T)$. Then from the semantics of the recursive operator, there are sequences $t_1,\ldots,t_m$ of triples in O^3 and $p_1,\ldots,p_m$ of triples in $e(T)$ such that $t_1 \in e(T)$, and $t_{m+1} = t_m \overset{i,j,k}{\underset{\theta,\eta}{\bowtie}} p_m$. If $m=1$ this follows from the first part of $\psi_{e'}$. If $m>1$, notice that, by definition, $I_T \models \varphi(t_j,t_{j+1})$ for each $1 \leq j < m$. It follows that $I_T \models \psi_{e'}$. The other direction is analogous.

The fact that the containment is strict follows from Part 3 of the proof.

Part 2 Next we prove that TrCl^3 is contained in TriAL^*. We do this by induction on TrCl^3 formulas. Note that all the cases, except for the case of transitive closure operator, are exactly the same as in the proof of Theorem 8.6.1. Next we show how to translate formulas of the form

$$\psi(x,y,z) := [\mathbf{trcl}_{x,y}\varphi(x,y,z)](u_1,u_2).$$

By the induction hypothesis there exists a TriAL^* expression R_φ such that for any triplestore T we have $I_T \models \varphi(a,b,c)$ iff $(a,b,c) \in R_\varphi(T)$.

Consider now the following expression R_ψ:

$$R := (R_\varphi \overset{1,2',3}{\underset{3=3' \wedge 2=1'}{\bowtie}})^*.$$

Observe now that a triple (a,b,c) will be contained in $R(T)$ iff there is a sequence of triples $(a,b_1,c),(b_1,b_2,c),(b_2,b_3,c),\ldots(b_k,b,c)$ with the property that they all belong to $R_\varphi(T)$. But this then means that the pair (a,b) belongs to the transitive closure of the relation defined by $\varphi(x,y,c)$. That is we have that $(a,b,c) \in R(T)$ iff b is reachable from a using only edges defined by $\varphi(x,y,c)$.

We now proceed case by case, depending on the structure of terms u_1 and u_2. Since our terms are only variables we have a total of nine cases.

- If $u_1 = x$ and $u_2 = y$ we define $R_\psi := R$. It is straightforward to see that $(a,b,c) \in R_\psi(T)$ iff $I_T \models \psi(a,b,c)$.
- If $u_1 = y$ and $u_2 = x$ we define $R_\psi := R$.
- If $u_1 = x$ and $u_2 = z$ we define $R_\psi := \sigma_{2=3}R$.
- If $u_1 = z$ and $u_2 = x$ we define $R_\psi := \sigma_{1=3}R$.
- If $u_1 = x$ and $u_2 = x$ we define $R_\psi := \sigma_{1=2}R$.
- All of the other cases are symmetric.

This concludes the proof in the case when φ above uses x,y,z as variables. All of the other cases are similar, e.g. when we have the formula $[\mathbf{trcl}_{x,y}\varphi(x,y,x)](x,y)$ the expression $(\sigma_{1=3}R_\varphi \bowtie^{1,2',3}_{2=1'})^*$ in place of R will suffice (note that now we have only two free variables).

That the containment is strict follows from the comments at the beginning of the proof of Part 3 below.

Part 3 We begin by showing that TriAL* is not contained in TrCl^4 or TrCl^5. In the proof of Theorem 8.6.1 we show that TriAL, and thus TriAL* contain an expression e such that $e(T)$ is nonempty if and only if T has 6 different objects. The proof then follows by two classical results in finite model theory [Libkin, 2004]: (1) e cannot be expressed by neither $\mathcal{L}^4_{\infty\omega}$ not $\mathcal{L}^5_{\infty\omega}$, the infinitary logic restricted to 4 and 5 variables, respectively, and (2) TrCl^k is contained in $\mathcal{L}^k_{\infty\omega}$

To see that TrCl^4 is not contained in TriAL (and thus that neither TrCl^5 not TrCl^6 are contained in TriAL), we define an analog of the logic FO^3-join used in the proof of Theorem 8.6.1. The logic FO$^3_\infty$-join extends FO^3-join with countably infinite disjunctions and conjunctions of formulas in FO^3-join (of course the restriction on the variables still holds). Formally, every FO^3-join formula is in FO$^3_\infty$-join, and if all φ_i are formulas in FO$^3_\infty$-join using the same set of at most 3 variables, for $i \in S$, where S is not necessarily finite, then $\bigwedge_{i\in S}\varphi_i$ and $\bigvee_{i\in S}\varphi_i$ are formulas in FO$^3_\infty$-join.

Notice that, by using these disjunctions, it is trivial to express the recursive star operator of TriAL* with FO$^3_\infty$-join. Thus, if two structures $\mathcal{A}$ and $\mathcal{B}$ are indistinguishable by FO$^3_\infty$-join, then so are they by TriAL*.

On the other hand, using the techniques in [Libkin, 2004] it is not difficult to see that, if two structures $\mathcal{A}$ and $\mathcal{B}$ are indistinguishable by FO$^3_\infty$-join iff they are indistinguishable by FO^3-join (if the spoiler can win the join game on $\mathcal{A}$ and $\mathcal{B}$, then it can win the infinitary join game that characterizes FO$^3_\infty$-join).

It follows from the above observations, and the proof of Theorem 8.6.1, that TriAL* cannot express the query

$$\varphi(x,y,z) = \exists x\exists y\exists z\exists w\big(\psi(x,y,w)\wedge\psi(x,w,z)\wedge\psi(w,y,z)\wedge\psi(x,y,z)\wedge x\neq y\wedge x\neq z\wedge x\neq w\wedge y\neq z\wedge y\neq w\wedge z\neq w\big),$$

where

$$\psi(x,y,z) = \exists w\big(E(x,w,y)\wedge E(y,w,x)\wedge E(y,w,z)\wedge E(x,w,y)\wedge E(x,w,z)\wedge E(z,w,x)\wedge x\neq z\wedge x\neq y\wedge y\neq z\big).$$

used in the proof of Theorem 8.6.1. □

8.7 Summary

In this chapter we have seen that although graph query languages form a good basis for navigational querying of RDF documents, certain properties of the model require a more general approach. Indeed, nested queries such as the one from Proposition 8.1.2 are often required in applications such as data integration, provenance tracking, or clustering, and the inherent inability of graph languages to deal with them becomes somewhat of an issue. Coding triples as graphs can be seen as one solution to this problem, however, this will not always work (without incurring a significant computational cost) and more organic languages, tailored specifically for RDF are required. To that end it is advantageous to recognize that reachability over graphs – binary in its essence – differs significantly from reachability over triples, where more general form of navigation is needed.

To overcome this issue we have proposed TriAL and TriAL*, languages designed to operate specifically over triples. Like relational algebra, taking relations as input and producing relations as output, we designed our language to be closed. Therefore a TriAL query will always produce a valid triplestore, not taking us outside of the studied model. Furthermore, the language was shown to be efficient, highly expressive and able to handle generalized reachability queries that fall out of scope of graph languages or SPARQL. The language also has a tidy declarative counterpart – a fragment of datalog called TripleDatalog$^\neg$, and is strongly rooted in logic. All of this seems to point to high potential applicability of the language, particularly taking into consideration that most of the features, namely joins, which form the crux of the language, have been implemented and optimized on all of the currently available RDBMSs. Of course, it remains to see if such systems can scalably implement the type of recursion we require, and to test how such an implementation stacks against currently used RDF systems.

Part III

Analysing the languages: Comparison and Containment

Chapter 9

Comparing the languages

In this chapter we compare previously introduced query languages in terms of expressive power. In particular we will present the complete picture of how the classes are related to each other and also examine purely navigational power of graph languages introduced in Part II. Note that navigational fragments of path queries from Part I collapse to RPQs and their relative expressiveness is well understood [Barceló, 2013].

As before, we will say that a language $\mathcal{L}_1$ is contained in a language $\mathcal{L}_2$ if for every query in $\mathcal{L}_1$ there is an equivalent query in $\mathcal{L}_2$. If in addition $\mathcal{L}_2$ has a query not expressible in $\mathcal{L}_1$, then $\mathcal{L}_1$ is strictly contained in $\mathcal{L}_2$. The languages are equivalent if each is contained in the other. They are incomparable if none is contained in the other.

We begin by comparing path languages to each other and show a strict hierarchy starting with RQDs and ending with RDPQs, with the exception of RQVs, which are, as established earlier, orthogonal to all of those. We then move onto GXPath and show that while the language is more expressive than *RQD*s, its inability to store data into variables makes it incomparable to other path languages. Note that here it also makes sense to study the expressive power of purely navigational language and compare it to that of NREs and CRPQs, since GXPath does allow some, albeit limited, amount of conjunction. Finally, we demonstrate how TriAL* can be used as a graph query language and show that, although it subsumes GXPath, it still has the same weakness of not being able to use variables, thus making it incomparable to RDPQs and other path formalisms that do have this functionality.

9.1 Path queries

From semantics of path queries in Chapter 4 it readily follows that a class of queries $\mathcal{L}_1$ is subsumed by $\mathcal{L}_2$ if and only if the class of automata of expressions used to define queries in $\mathcal{L}_2$ are more expressive than the ones defining $\mathcal{L}_1$. To that end it suffices to compare language theoretic formalisms defining path queries to gauge their relative expressive power. It is also

easy to see that whether we consider languages over data words or over data paths has no impact on the final result (see Section 3.1).

Taking this into consideration, applying Theorem 6.6.1 immediately implies the following set of results.

Theorem 9.1.1. *The following relations hold, where $\subsetneq$ denotes that language on the left is subsumed by the language on the right, but not vice versa.*

- *RQDs* $\subsetneq$ *RQBs* $\subsetneq$ *RQMs* $=$ RDPQ*s.*
- *RQVs are incomparable in terms of expressive power with RQDs , RQBs , RQMs and* RDPQ*s.*

9.2 Moving up the food chain

Here we compare GXPath to path languages introduced in Chapter 4 as well as to traditional navigational languages such as RPQs, CRPQs and NREs. Note that GXPath enriches RPQs with new navigational abilities and it is therefore worthwhile examining how navigational part of the language fares when compared to other extensions of RPQs.

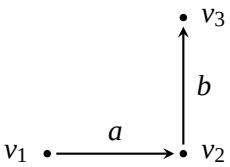
GXPath and path languages When comparing GXPath with path languages we will consider the regular fragment with $\sim$ type data tests, since they subsume classical XPath-style tests. While it is apparent from the definition of $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\sim)$ that it contains *RQD*s, we can also show that the containment is strict.

Proposition 9.2.1. *The class of RQD queries is strictly contained in* $\mathsf{GXPath}_{reg}(\mathsf{c},\sim)$.

Proof. To see that the containment is strict consider the following GXPath query:

$$q = (a[b])^*.$$

Note that this is also an NRE. To obtain a contradiction assume that there is some *RQD* Q_q equivalent to q. Now consider the following graph G.



Data values are not important here so we do not list them explicitly. It is easily checked that $(v_1, v_2) \in [\![q]\!]^G$. By our assumption we also have that $(v_1, v_2) \in Q_q(G)$. But since Q_q is an *RQD* this means that there is some regular expression with equality e_q such that $Q_q = x \xrightarrow{e_q} y$ and:

- There is a path π starting with v_1 and ending with v_2, and
- $\lambda(\pi)$ belongs to $\mathcal{L}(e_q)$.

However, the only path in G connecting v_1 and v_2 is $\pi = v_1 a v_2$. Consider now the graph G' obtained from G by removing the edge (v_2, b, v_3). We now have $(v_1, v_2) \notin [\![q]\!]^{G'}$, but $\pi = v_1 a v_2$ is still a path in G' with $\lambda(\pi) \in \mathcal{L}(e_q)$. This then implies that $(v_1, v_2) \in Q_q(G')$, a contradiction. □

Comparing GXPath to more expressive path languages we can see that the ability to use variables makes them capable of expressing queries outside the reach of GXPath. We also show that the converse is true, as new navigational features allow GXPath to define patterns not captured by paths.

Proposition 9.2.2. *GXPath$_{reg}(c, \sim)$ is incomparable in terms of expressive power with RQMs, RQBs,* RDPQ*s and RQVs.*

Proof. It is easily seen that the example from Proposition 9.2.1 can be used to give a GXPath query not expressible by any of the path languages.

To prove the reverse we show that GXPath$_{\text{reg}}(c, \sim)$ is contained in three variable infinitary logic $\mathcal{L}^3_{\infty\omega}$ (with constants and data value comparisons). It is well know that this logic can not define models that have at least four different elements [Libkin, 2004]. However, one can readily check that such a query is expressible by any of the path formalisms mentioned in this theorem. We will give a full proof of this fact for a slightly stronger class of queries in Theorem 9.3.8. This, together with Proposition 9.3.6, implies the desired result. □

Relative expressiveness of navigational fragments Our next goal is to compare the expressiveness of navigational GXPath fragments with that of traditional graph languages. We start with *nested regular expressions*, and after that look at path languages such as RPQs, CRPQs, and relatives.

As expected, GXPath$_{\text{reg}}$ is strictly more expressive than NREs. However, we show that NREs do capture the positive fragment of GXPath$_{\text{reg}}$.

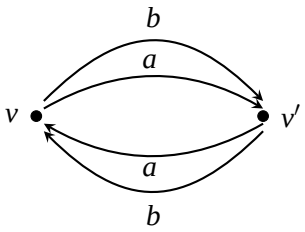
Theorem 9.2.3. *GXPath$_{reg}^{pos}$ = NRE $\subsetneq$ GXPath$_{reg}^{path\text{-}pos}$.*

Proof. First we show that NRE $\subsetneq$ GXPath$_{\text{reg}}$.

Using a straightforward inductive construction one can show how to convert a nested regular expression into an equivalent path expression of GXPath$_{\text{reg}}$. Note that all the operations can be written down verbatim, minus the $[n]$ expression whose GXPath$_{\text{reg}}$ equivalent is $[\langle e_n \rangle]$, where e_n is an expression equivalent to n.

Next we show that GXPath$_{\text{core}}$ query $q = a[\neg\langle b \rangle]$ is not expressible by any NRE.

Consider the following data graph G.



It is easy to see that $[\![q]\!]^G = \emptyset$. We now show that $[\![n]\!]^G \neq \emptyset$, for any nested regular expression n. Thus we conclude that no equivalent NRE exists.

In fact we show that for every NRE n there exist nodes $x_1, x_2, y_1, y_2 \in \{v, v'\}$ such that $(v, x_1), (v', x_2), (y_1, v), (y_2, v') \in [\![n]\!]^G$.

This can be shown by an easy induction on the structure of n.

We now show that $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}} = \mathsf{NRE}$.

We already know that nested regular expressions can be expressed as GXPath queries. Examining the proof shows us that no negation is needed for this.

To complete the proof we now show how to convert any $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$ expression into an equivalent nested regular expression. More precisely, we show that for any path expression α of our fragment there exists a nested regular expression n_α such that for any graph G we have $(x,y) \in [\![\alpha]\!]^G$ iff $(x,y) \in [\![n_\alpha]\!]^G$. Moreover, for any node expression φ we define a nested regular expression n_φ such that $x \in [\![\varphi]\!]^G$ iff $(x,x) \in [\![n_\varphi]\!]^G$. We do this by induction on the structure of our $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$ expressions.

Basis:

- $e = a$ then $n_e = a$
- $e = a^-$ then $n_e = a^-$
- $e = \varepsilon$ then $n_e = \varepsilon$
- $e = \top$ then $n_e = \varepsilon$

Inductive step:

- $e = [\varphi]$ then $n_e = [n_\varphi]$
- $e = \alpha \cdot \beta$ then $n_e = n_\alpha \cdot n_\beta$
- $e = \alpha \cup \beta$ then $n_e = n_\alpha + n_\beta$
- $e = \varphi \wedge \psi$ then $n_e = \varepsilon[n_\varphi] \cdot \varepsilon[n_\psi]$
- $e = \varphi \vee \psi$ then $n_e = \varepsilon[n_\varphi + n_\psi]$
- $e = \langle \alpha \rangle$ then $n_e = \varepsilon[n_\alpha]$.

It is easy to see the equivalence between defined expressions. □

We will now show that XPath-like formalisms are incomparable with CRPQs and similar queries in terms of their navigational expressiveness. The simple restriction, $\mathsf{GXPath}_{\text{reg}}^{\text{pos}}$, is not subsumed by CRPQs. In fact it is not even subsumed by unions of two-way CRPQs (which allow navigation in both ways). On the other hand, CRPQs are not subsumed by the strongest of our navigational languages, $\mathsf{GXPath}_{\text{reg}}$.

Theorem 9.2.4. *CRPQs and GXPath fragments are incomparable:*

- $\mathit{GXPath}_{reg}^{pos} \not\subseteq \mathit{CRPQ}$ *(even stronger, there are* $\mathit{GXPath}_{reg}^{pos}$ *queries not definable by U2CRPQs);*
- $\mathit{CRPQ} \not\subseteq \mathit{GXPath}_{reg}$.

Proof. Note that the first item follows from Theorem 9.2.3 and Theorem 1 in [Barceló et al., 2012c].

To see that the second item holds we first show that for every $\mathsf{GXPath}_{\text{reg}}$ expression e there exists an $\mathcal{L}^3_{\infty\omega}$ formula F_e equivalent to it. After that we give an example of a CRPQ that is not expressible in this logic using a standard multi-pebble games argument.

To be more precise we will be working with $\mathcal{L}^3_{\infty\omega}$ formulas over the alphabet $\{E_a : a \in \Sigma\}$ (and with the equality symbol). All the relations are binary and simply represent a labeled edge between two nodes. We will denote data graphs as structures for this logic by $G = \langle V, (E_a)_{a\in A}, = \rangle$.

Now for every path expression α we will define a formula $F_\alpha(x,y)$ such that $(v,v') \in [\![\alpha]\!]^G$ iff $G \models F_\alpha[x/v, y/v']$. Likewise for a node expression φ we define a formula $F_\varphi(x)$ such that $v \in [\![\varphi]\!]^G$ iff $G \models F_\varphi[x/v]$.

We do this by induction on $\mathsf{GXPath}_{\text{reg}}$ expressions.

Basis:

- $\alpha = a$ then $F_\alpha(x,y) \equiv E_a(x,y)$
- $\alpha = a^-$ then $F_\alpha(x,y) \equiv E_a(y,x)$
- $\alpha = \varepsilon$ then $F_\alpha(x,y) \equiv x = y$
- $\varphi = \top$ then $F_\alpha(x) \equiv x = x$

Inductive step:

- $\alpha' = [\varphi]$ then $F_{\alpha'}(x,y) \equiv x = y \wedge F_\varphi(x)$
- $\alpha' = \alpha \cdot \beta$ then $F_{\alpha'}(x,y) \equiv \exists z(\exists y\ (y = z \wedge F_\alpha(x,y)) \wedge \exists x\ (x = z \wedge F_\beta(x,y)))$
- $\alpha' = \alpha \cup \beta$ then $F_{\alpha'}(x,y) \equiv F_\alpha(x,y) \vee F_\beta(x,y)$
- $\alpha' = \alpha^*$ then define
  - $\varphi^1_\alpha(x,y) \equiv F_\alpha(x,y)$,

  - $\varphi_\alpha^{n+1}(x,y) \equiv \exists z\,(\exists y\,(y = z \wedge F_\alpha(x,y)) \wedge \exists x\,(x = z \wedge \varphi_\alpha^n(x,y)))$
  - Finally, set $F_{\alpha'}(x,y) \equiv \bigvee_{n\in\omega} \varphi_\alpha^n(x,y)$

- $\alpha' = \overline{\alpha}$ then $F_{\alpha'}(x,y) \equiv \neg F_\alpha(x,y)$
- $\varphi' = \neg\varphi$ then $F_{\varphi'}(x) \equiv \neg F_\varphi(x)$
- $\varphi' = \varphi \wedge \varphi$ then $F_{\varphi'}(x) \equiv F_\varphi(x) \wedge F_\psi(x)$
- $\varphi' = \langle\alpha\rangle$ then $F_{\varphi'}(x) \equiv \exists y F_\alpha(x,y)$.

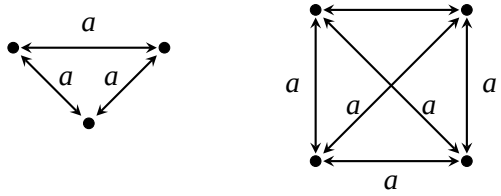
It is straightforward to show that the translation has the desired property.

Next we define a binary CRPQ $\varphi(x,y)$ that has no $\mathsf{GXPath}_{\mathrm{reg}}$ equivalent.

$$\begin{aligned}\varphi(x,y) := (x,a,y) \wedge (x,a,z) \wedge (x,a,w) \wedge \\ (y,a,x) \wedge (z,a,x) \wedge (w,a,x) \wedge \\ (y,a,z) \wedge (y,a,w) \wedge \\ (z,a,y) \wedge (w,a,y) \wedge \\ (z,a,w) \wedge (w,a,z).\end{aligned}$$

Note that φ is stating that our graph has a complete subgraph of size four.

Next we take two graphs G_1 and G_2 as in the following figure.



G_1 G_2

Note that G_1 is a complete graph of three vertices with all the edges labeled a and G_2 is the same, but with four vertices. It is straightforward to see that $\varphi(G_1) = \emptyset$, while $\varphi(G_2) \neq \emptyset$.

It is well known that no $\mathcal{L}^3_{\infty\omega}$ sentence F can distinguish the two models (see, e.g., [Libkin, 2004]). This is due to the fact that that duplicator has a winning strategy in an infinite 3-pebble game on these graphs, simply by preserving equality of pebbled elements. That is for any F we have $G_1 \models F$ iff $G_2 \models F$. Note that our result follows, since the above CRPQ selects the entire graph on G_2 and the empty graph on G_1. This completes our proof. □

On the other hand, the positive fragment of $\mathsf{GXPath}_{\mathrm{core}}$ can be captured by unions of two-way CRPQs.

Proposition 9.2.5. *$\mathsf{GXPath}^{pos}_{core} \subsetneq$ U2CRPQ.*

Proof. From the previous theorem we know that there is a CRPQ not expressible in GXPath_{reg}.

On the other hand, for any $\mathsf{GXPath}^{pos}_{core}$ expression e we can construct an equivalent U2CRPQ. That is, for any path expression α we define a U2CRPQ, named $\psi_\alpha(x,y)$, in two free variables, x and y, such that for any graph database G we have $[\![\alpha]\!]^G = \psi_\alpha(G)$. Similarly for any node expression φ we define a U2CRPQ $\psi_\varphi(x)$. We do so by induction on the structure of $\mathsf{GXPath}^{pos}_{core}$ expressions.

Basis:

- For $\alpha = \varepsilon$ we have $\psi_\alpha(x,y) := (x,\varepsilon,y)$.
- For $\alpha = \_$ we have $\psi_\alpha(x,y) := \bigvee_{a\in\Sigma}(x,a,y)$.
- For $\alpha = a$ we have $\psi_\alpha(x,y) := (x,a,y)$.
- For $\alpha = a^-$ we have $\psi_\alpha(x,y) := (x,a^-,y)$.
- For $\alpha = a^*$ we have $\psi_\alpha(x,y) := (x,a^*,y)$.
- For $\alpha = a^{-*}$ we have $\psi_\alpha(x,y) := (x,a^{-*},y)$.
- For $\varphi = \top$ we have $\psi_\varphi(x) := \exists y(x,\varepsilon,y)$.

Inductive step:

- For $\alpha = [\varphi]$ we have $\psi_\alpha(x,y) := (x,\varepsilon,y) \wedge \psi_\varphi(y)$.
- For $\alpha = \alpha' \cdot \beta'$ we have $\psi_\alpha(x,y) := \exists z \psi_{\alpha'}(x,z) \wedge \psi_{\beta'}(z,y)$.
- For $\alpha = \alpha' \cup \beta'$ we have $\psi_\alpha(x,y) := \psi_{\alpha'}(x,y) \vee \psi_{\beta'}(x,y)$.
- For $\varphi = \varphi_1 \wedge \varphi_2$ we have $\psi_\varphi(x) := \psi_{\varphi_1}(x) \wedge \psi_{\varphi_2}(x)$.
- For $\varphi = \varphi_1 \vee \varphi_2$ we have $\psi_\varphi(x) := \psi_{\varphi_1}(x) \vee \psi_{\varphi_2}(x)$.
- For $\varphi = \langle\alpha\rangle$ we have $\psi_\varphi(x) := \exists y \psi_\alpha(x,y)$.

It is straightforward to show that the defined expressions are equivalent. □

9.3 Triple algebra and graph languages

Although introduced as a querying mechanism for RDF Triplestores, TriAL* can also be used to query graph databases. The goal of this section is to demonstrate how this can be achieved, both when considering graphs with or without data values and to show that TriAL* can be viewed as a natural extension of GXPath, allowing more involved types of queries and data tests. Since the language has not been studied in the graph context before, we will start by comparing it to traditional navigational languages and purely navigational fragments of GXPath before moving onto languages that handle data values.

Navigational graph query languages and TriAL* Here we compare TriAL* with a number of established formalisms for graph databases such as NREs, RPQs and *conjunctive* regular path queries (CRPQs). As our yardstick language for comparison we use GXPath$_{\text{reg}}$ which is essentially PDL [Harel et al., 2000]. Note that all of the navigational languages we consider here are designed to query the topology of a graph database and specify various reachability patterns between nodes. As such, they are naturally equipped with the star operator and to make our comparison fair we will compare them with TriAL* and not with TriAL.

Since TriAL* is designed to query triplestores, we need to explain how to compare its power with that of graph query languages. Given a graph database $G = (V, E)$ over the alphabet Σ, we define a triplestore $T_G = (O, E)$, with $O = V \cup \Sigma$. Note that for now we deal with navigation; later we shall also look at data values.

To compare TriAL* with binary graph queries in a graph query language $\mathcal{L}$, we turn TriAL* ternary queries Q into binary by applying the $\pi_{1,3}(Q)$, i.e., keeping (s, o) from every triple (s, p, o) returned by Q. Under these conventions, we say that a graph query language $\mathcal{L}$ is contained in TriAL* if for every binary query $\alpha \in \mathcal{L}$ there is a TriAL* expression e_α so that $\pi_{1,3}(e_\alpha)$ and α are equivalent, and likewise, TriAL* is contained in a graph query language $\mathcal{L}$ if for every expression e in TriAL* there is a binary query $\alpha_e \in \mathcal{L}$ that is equivalent to $\pi_{1,3}(e)$. The notions of being strictly contained and incomparable extend in the same way.

Alternatively, one can do comparisons using triplestores represented as graph databases, as in Proposition 8.1.2. Since here we study the ability of TriAL* to serve as a graph query language, the comparison explained above looks more natural, but in fact all the results remain true even if we do the comparison over triplestores represented as graph databases, as described in Section 8.1.

We now show that all GXPath$_{\text{reg}}$ queries can be defined in TriAL*, but that there are certain properties that TriAL* can define that lie beyond the reach of GXPath$_{\text{reg}}$.

Theorem 9.3.1. *GXPath$_{reg}$ is strictly contained in* TriAL*.

Proof. Assume that GXPath$_{\text{reg}}$ uses a finite alphabet Σ of labels. We show that GXPath$_{\text{reg}}$ is contained in TriAL* by simultaneous induction on the structure of GXPath$_{\text{reg}}$ expressions. If we are dealing with a path expression α we will denote the TriAL* expression equivalent to α by E_α. Similarly when dealing with node expression φ, the corresponding TriAL* expression will be denoted E_φ. Note that for the node expression φ of GXPath$_{\text{reg}}$ we consider the TriAL* expression E_φ to be its equivalent if the answer set of φ is the same as the answer of $\pi_1(E_\varphi)$ over all graph databases and their triplestore representations, respectively.

Through the proof we will make use of the universal relation U containing all possible combinations of elements present in the model. We will also make use of the diagonal relation $D = U \bowtie^{1,1,1}_{1=1} U$ selecting all the triples (a, a, a) with $a \in V$.

Basis:

- $\alpha = a$ then $E_\alpha = E \bowtie^{1,2,3}_{2=a} E$
- $\alpha = a^-$ then $E_\alpha = E \bowtie^{3,2,1}_{2=a} E$
- $\alpha = \varepsilon$ then $E_\alpha = U \bowtie^{1,1,1}_{1=1} U$
- $\varphi = \top$ then $E_\varphi = U \bowtie^{1,1,1}_{1=1} U$

Inductive step:

- $\alpha' = [\varphi]$ then $E_{\alpha'} = E_\varphi \bowtie^{1,1,1}_{1=1} E_\varphi$
- $\alpha' = \alpha \cdot \beta$ then $E_{\alpha'} = E_\alpha \bowtie^{1,2,3'}_{3=1'} E_\beta$
- $\alpha' = \alpha \cup \beta$ then $E_{\alpha'}(x,y) = E_\alpha \cup E_\beta$
- $\alpha' = \alpha^*$ then $E_{\alpha'} = (E_\alpha \bowtie^{1,2,3'}_{3=1'})^*$
- $\alpha' = \overline{\alpha}$ then $E_{\alpha'} = E^c_\alpha$
- $\varphi' = \neg\varphi$ then $E_{\varphi'} = E^c_\varphi \cap D$
- $\varphi' = \varphi \wedge \varphi$ then $E_{\varphi'} = E_\varphi \cap E_\psi$
- $\varphi' = \langle\alpha\rangle$ then $E_{\varphi'} = E_\alpha \bowtie^{1,1,1}_{1=1} E_\alpha$.

It is straightforward to check that this translation works as intended. For illustration, consider the case when $\alpha' = \alpha \cdot \beta$. Our induction hypothesis is that we have two expressions, E_α and E_β such that (a,b) is in the answer to α on G iff $(a,c,b) \in E_\alpha(T_G)$, for some c and similarly for β. Assume now that (a,b) is in the answer to α' on G. Then there is c such that (a,c) is in the answer to α and (c,b) in the answer to β. But then $(a,c',c) \in E_\alpha(T_G)$ and $(c,b',b) \in E_\beta(T_G)$ for some c',b'. By the definition of join, we conclude that $(a,c',b) \in E_{\alpha'}(T_G)$. Note that all the implications above were in fact equivalences, so we get the opposite direction as well. All of the other cases follow similarly.

To show that the containment is strict recall that in Theorem 9.2.4 we proved that $\mathsf{GXPath}_{\mathrm{reg}}$ is contained in $\mathcal{L}^3_{\infty,\omega}$. Consider now the following TriAL expression:

$$U \overset{1,2,3}{\underset{\varphi}{\bowtie}} U,$$

where $\varphi = (1 \neq 2) \wedge (1 \neq 3) \wedge (1 \neq 1') \wedge (2 \neq 3) \wedge (2 \neq 1') \wedge (3 \neq 1') \wedge \bigwedge_{a \in \Sigma, 1 \leq i \leq 3} i \neq a \wedge \bigwedge_{a \in \Sigma, 1' \leq i \leq 3'} i \neq a$ and U is the universal relation. It follows easily that this expression has an nonempty answer set if and only if the original graph database had at least four different nodes. It is well known that this query is not expressible in $\mathcal{L}^3_{\infty,\omega}$, thus implying that the containment is indeed strict. □

Recall from Theorem 9.2.3 that $\mathsf{GXPath}_{\mathrm{reg}}$ subsumes NREs. Thus:

Corollary 9.3.2.

- *NREs are strictly contained in* TriAL*.
- *RPQs are strictly contained in* TriAL*.

Next we move to comparison with conjunctive queries. Here, instead of usual CRPQs we will consider slightly more expressive conjunctive NREs (CNREs) [Barceló et al., 2013a]. Formally, these are expressions of the form $\varphi(\overline{x}) = \exists \overline{y} \bigwedge_{i=1}^{n}(x_i \xrightarrow{e_i} y_i)$, where all variables x_i, y_i come from $\overline{x}, \overline{y}$ and each e_i is a NRE. The semantics extends that of NREs, with each $x_i \xrightarrow{e_i} y_i$ interpreted as the existence of a pattern between them that is denoted by e_i. We compare TriAL* with these queries, and also with *unions* of CNREs that use bounded number of variables.

In order to do these comparisons we will rely on the fact that TriAL* is subsumed by infinitary logic with six variables.

Lemma 9.3.3. TriAL* *is contained in the infinitary logic* $\mathcal{L}^6_{\infty,\omega}$.

Proof. What we mean by this is along the lines of the proof of Theorem 8.6.1 (Part 1), where we compare TriAL with first-order logic over the vocabulary $(E_1, \ldots, E_l, \sim)$.

That is to prove the lemma, we only have to show that the * operator can be simulated in this logic. To see this consider an arbitrary star-join of the form

$$R = (F \overset{i',j',k'}{\underset{\theta,\eta}{\bowtie}})^*.$$

Assume that we have an $\mathcal{L}^6_{\infty,\omega}$ formula $F(x_1,x_2,x_3)$ such that $T \models F(a,b,c)$ if and only if $(a,b,c) \in F(T)$. We first define the following formulas α, β. Consider the formula θ. We then let α be the conjunctions of formulas $x_i = x_j$, whenever $i = j$ is a conjunct in θ and $x_i \neq x_j$, whenever $i \neq j$ is a conjunct in θ. Similarly for $\rho(i) = \rho(j)$ in η we add $x_i \sim x_j$ as a conjunct in β and analogously for $\rho(i) \neq \rho(j)$.

We now define the following formulas:

- $R_1(x_1,x_2,x_3) := F(x_1,x_2,x_3)$.
- $R_{n+1}(x_1,x_2,x_3) := \exists x_4,x_5,x_6(R_n(x_1,x_2,x_3) \wedge \alpha \wedge \beta \wedge \exists x_1,x_2,x_3(x_4 = x_1 \wedge x_5 = x_2 \wedge x_6 = x_3 \wedge F(x_1,x_2,x_3)))$

Finally set $R(x_1,x_2,x_3) := \bigvee_{n\in\omega} R_n(x_1,x_2,x_3)$.

It is straightforward to check that this formula defines the desired relation over T. A similar formula can be defined for left-joins.

Note that we could have included constants to our comparisons with FO, but to keep the language one-sorted we omit them from our presentation. It is a straightforward exercise to

check that all of the results would still hold true is they were allowed. For example constant comparisons of the form $2 = a$ would be handled by adding the clause $x_2 = a$ as a conjunct to the formula α above. □

When comparing TriAL* with CNREs we obtain the following.

Theorem 9.3.4.

- *CNREs and* TriAL* *are incomparable in terms of expressive power.*
- *Unions of CNREs that use only three variables are strictly contained in* TriAL*.

Proof. We begin by proving that full CNREs and TriAL* are incomparable in terms of expressive power.

The existence of a CNRE query not expressible by TriAL* simply follows from the fact that TriAL* is contained in $\mathcal{L}^6_{\infty,\omega}$. The reason for this is that CNREs can ask for a 7-clique, a property not expressible in $\mathcal{L}^6_{\infty,\omega}$.

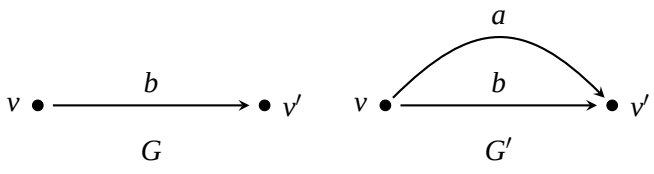
To see the reverse we will use a well know fact that CNREs are a monotonic class of queries. That is for any two graph databases G and G' such that $G \subseteq G'$ (that is G' contains all the nodes and edges of G) and any CNRE q we have that (u,v) is in the answer to q on G implies that (u,v) is in the answer to q on G' as well.

Next consider TriAL expression

$$e := (E \underset{2=a}{\overset{1,2,3}{\bowtie}} U)^c \underset{\varphi}{\overset{1,2,3}{\bowtie}} U,$$

with $\varphi = \bigwedge_{b\in\Sigma} 1 \neq b, 3 \neq b$. When interpreted over (a translation into a triplestore of) a graph database G, this expression returns all pairs of nodes that are *not* connected by an a-labeled edge. (Formally we will return all the triples u,v,w such that u and w are not connected by an a-labeled edge. The extra join just handles the specifics of our translation of a graph database into a triplestore). Suppose now that there is a CNRE q defining the aforementioned query.

Consider the following two graphs.



The nodes (v,v') will be in the answer to our query over the graph G. Using the monotonicity of CNREs and the fact that G is contained in G' we conclude that (v,v') is also in the answer to our query over G'. Note that this is a contradiction since we assumed that q extracts all pairs of nodes not connected by an a-labeled path.

This concludes the proof of part one of our Theorem.

Next we show that UCNREs using only three distinct variables are contained in TriAL*. Observe first that for any NRE e there is a TriAL* expression E_e equivalent to e over all data graphs (Corollary 9.3.2). We will now show that any CNRE that uses precisely three variables is definable using TriAL. To see this, consider the following example. Let Q be the following CNRE:

$$Q(x,y,z) := (x,e_1,y) \wedge (z,e_2,y) \wedge (y,e_3,y) \wedge (y,e_4,x).$$

It is easy to check that the following TriAL expression:

$$(((T_{e_1} \underset{1=1}{\overset{1,2,3}{\bowtie}} U) \underset{2=2'}{\overset{1,2,3}{\bowtie}} (T_{e_2} \underset{1=1}{\overset{1,3,2}{\bowtie}} U)) \underset{2=2'}{\overset{1,2,3}{\bowtie}} (T_{e_3} \underset{1=3}{\overset{2,1,2}{\bowtie}} U)) \underset{2=2',1=1'}{\overset{1,2,3}{\bowtie}} (T_{e_3} \underset{1=1}{\overset{3,1,2}{\bowtie}} U),$$

where T_{e_i} is the TriAL equivalent of e_i, is equivalent to Q over all graph databases.

Notice that here we have to output all the triples (x,y,z) satisfying the condition of our conjunctive query. For this we first join each T_{e_i} with the universal relation and arrange the nodes potentially appearing in the answer in the right order. For example, when dealing with (x,e_1,y) we define $T_{e_1} \bowtie^{1,2,3}_{1=1} U$, where we put the nodes appearing in T_{e_1} in the correct order. At the end we simply join all the resulting relation in a way that preserves the designated objects. Here we have to take care that we force equality only on the objects used in the conjunctions involved up to now.

It is straightforward to extend this construction to the most general case of an arbitrary number of conjuncts with various arrangement of variables.

Finally, since TriAL expressions are closed under union we get that UCNREs with only three variables are contained in TriAL*. That the containment is proper follows from the first part of the proof. □

By observing that the expressions separating CNREs from TriAL* are CRPQs, and that CNREs are more expressive than CRPQs and C2RPQS [Barceló et al., 2012c] we obtain:

Corollary 9.3.5.

- *CRPQs and* TriAL* *are incomparable in terms of expressive power.*
- *Unions of C2RPQs and CRPQs that use only three variables are strictly contained in* TriAL*.

Data values in TriAL* Until now we have compared our algebra with purely navigational formalisms. Triple stores do have data values, however, and can thus model any graph database. That is, for any graph database $G = (V,E,\rho)$ we can define a triplestore $T_G = (O,E,\rho)$ with $O = V \cup \Sigma$. Note that nodes corresponding to labels have no data values assigned in our model. This is not an obstacle and can in fact be used to model graph databases that have data values on both the nodes and the edges.

To compare $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\sim)$ with TriAL^*, we use the same convention as for navigational languages.

Proposition 9.3.6. *$GXPath_{reg}(c,\sim)$ is strictly contained in* TriAL^*.

Proof. The proof here follows the same lines as the one of Theorem 9.3.1. Because of this we only have to show how to define an equivalent TriAL^* expression for any of the newly added data operators in $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\sim)$.

- For $\varphi = \langle \alpha = \beta \rangle$ we define $E_\varphi = E_\alpha \bowtie^{1,1,1}_{1=1',\rho(3)=\rho(3')} E_\beta$
- For $\varphi = \langle \alpha \neq \beta \rangle$ we define $E_\varphi = E_\alpha \bowtie^{1,1,1}_{1=1',\rho(3)\neq\rho(3')} E_\beta$
- For $\alpha' = \alpha_=$ we define $E_{\alpha'} = E_\alpha \bowtie^{1,2,3}_{\rho(1)=\rho(3)} E_\alpha$
- For $\alpha' = \alpha_\neq$ we define $E_{\alpha'} = E_\alpha \bowtie^{1,2,3}_{\rho(1)\neq\rho(3)} E_\alpha$
- For $\varphi = (= c)$, with c a constant, we put $E_\varphi = U \bowtie^{1,1,1}_{1=1',\rho(1)=c} U$, where U is the universal relation introduced previously.

It is again straightforward to see that the described translations works as desired.

To show that the containment is strict we use a similar approach as when proving Theorem 9.3.1. We first notice that the proof of Theorem 9.2.4 can easily be extended to show that $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c},\sim)$ is subsumed by $\mathcal{L}^3_{\infty,\omega}(\sim)$, the infinitary three variable logic with data value tests. Here the only addition to the logic is the ability to use formulas of the form $x \sim y$ that are true if and only if x and y have the same data value.

More formally, we will represent a data graph $G = (V, E, \rho)$ as a FO structure $G = (V, (E_a : a \in \Sigma), \sim)$ with $E_a = \{(v, v') : (v, a, v') \in E\}$. It is straightforward to see that with this interpretation we have $\mathsf{GXPath}_{\mathrm{reg}}(\sim) \subseteq \mathcal{L}^3_{\infty,\omega}(\sim)$. Constants can be added in a straightforward way.

It is also easy to see that the 3-pebble game [Libkin, 2004] for $\mathcal{L}^3_{\infty,\omega}(\sim)$ follows the intended semantics when interpreted over data graphs. (Note that the game works over any class of structures, but over data graphs only relations are edge relations and the data value comparison.)

We can now play the 3-pebble game over the 3-clique graph and the 4-clique graph [Libkin, 2004] where all data values are the same. The same winning strategy for the duplicator as in the game with no data values will still work, so we conclude that $\mathcal{L}^3_{\infty,\omega}(\sim)$ can not distinguish the two models.

Consider now the following TriAL expression:

$$U \overset{1,2,3}{\underset{\varphi}{\bowtie}} U,$$

where $\varphi = (1 \neq 2) \wedge (1 \neq 3) \wedge (1 \neq 1') \wedge (2 \neq 3) \wedge (2 \neq 1') \wedge (3 \neq 1') \wedge \bigwedge_{a\in\Sigma, 1\leq i\leq 3} i \neq a \wedge \bigwedge_{a\in\Sigma, 1'\leq i\leq 3'} i \neq a$ and U is the universal relation. It follows easily that this expression has different answer on the two models (since it asks for four different nodes in the original graph database). This finishes our proof. □

This also implies that TriAL* subsumes *RQD*s.

Corollary 9.3.7. *The class of RQD queries is strictly contained in* TriAL*.

Finally, we compare TriAL* with path languages that use variables to store data.

Proposition 9.3.8. TriAL* *is incomparable in terms of expressive power with* RDPQ*s, RQMs, RQBs and RQVs.*

Proof. We begin by showing that *RQM*s are not contained in TriAL*. To see this recall from Lemma 9.3.3 that TriAL* is subsumed by infinitary logic $\mathcal{L}^6_{\infty,\omega}$.

Next we observe that for any n *RQM*s can define a property not expressible in $\mathcal{L}^n_{\infty,\omega}$. For this consider the following regular expression with memory:

$e_2 := \downarrow x_1 a[x_1^{\neq}] \downarrow x_2$

$e_{n+1} := e_n \cdot a[x_1^{\neq} \wedge x_2^{\neq} \wedge \cdots \wedge x_n^{\neq}] \downarrow x_{n+1}$.

Since no node can have more than one data value attached it follows that the answer to the query posted by the expression e_n is nonempty if and only if the graph database has at least n different elements.

It is well known [Libkin, 2004] that $\mathcal{L}^n_{\infty,\omega}$ can not define a query stating that the model has at least $n+1$ element. Since TriAL* is contained in $\mathcal{L}^6_{\infty,\omega}$ the desired result follows from the fact that e_7 is nonempty only on the graphs with at least 7 elements. Observe now that the expressions used here are in fact regular expressions with binding and it is easily checked that the same language can be defined by variable automata.

To show that there are TriAL* queries outside of reach of path languages from Chapter 4, recall that TriAL* subsumes $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{c}, \sim)$ (Theorem 9.3.1) and the later already has the required property (Proposition 9.2.2). □

9.4 The complete picture

Having compared data graph languages we can see that different data manipulation abilities not only make the complexity of query evaluation significantly different, but also have a big impact on the type of queries they are capable of expressing. For example the ability to use variables allows path languages to express queries outside of the scope of navigationally richer languages like GXPath and TriAL*, which do come with the ability to manipulate objects as e.g. logic does, but only using a fixed amount of variables. On the other hand the ability of graph languages to express various navigational patterns places them outside of reach of any path language, since these languages can not go beyond RPQs in their ability to specify how nodes in the graph are connected. Furthermore, we can establish a strict hierarchy amongst path languages, starting with *RQD*s and ending with RDPQs and their expression equivalent *RQM*s, with the exception of *RQV*s. In fact, we saw that the somewhat unnatural capability of

variable automata to reason about paths non-locally makes the class of *RQV* queries orthogonal to all other languages introduced in previous chapters. Summary of all of the results is given in Figure 9.1.

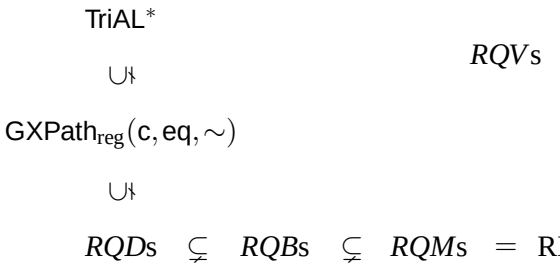


Figure 9.1: Comparison of data graph languages. Lack of a $(\subsetneq_{\neq} + =)^*$ labelled path between two languages signifies that they are incomparable.

Chapter 10

Query containment

The goal of this chapter is to initiate the study of static analysis aspects of graph query languages. In what follows we will concentrate on the query containment problem, which is the problem of deciding, given two queries in some graph language, whether the answer set of the first query is contained in the answer set of the second one. Deciding query containment is a fundamental problem in database theory, and is relevant to several complex database tasks such as data integration [Lenzerini, 2002], query optimisation [Abiteboul et al., 1995], view definition and maintenance [Gupta and Mumick, 1995], and query answering using views [Calvanese et al., 2001].

The importance of this problem has motivated sustained research for relational query languages (see e.g. [Abiteboul et al., 1995]), XML query languages (see e.g. [Schwentick, 2004]) and even extensions of RPQs and other graph query languages [Barceló et al., 2012b, Barceló et al., 2011, Calvanese et al., 2000, Florescu et al., 1998]. The overall conclusion is that containment is generally undecidable for first order logic and other similar formalisms (see e.g. [Abiteboul et al., 1995]), but becomes decidable if we restrict to queries with little or no negation. For example, containment of conjunctive queries is NP-complete, while containment of RPQs, 2-way RPQs and nested regular expressions is PSPACE-complete. For CRPQs it jumps to EXPSPACE-complete.

While much is known about the containment of above mentioned classes of queries, containment for languages with data value comparisons has only been looked at recently in [Kostylev et al., 2014]. Here we extend that work to include all of the query classes introduced in the previous sections. In what follows we primarily concentrate on containment, but the techniques used can easily be adapted to deal with other similar problems, such as satisfiability or equivalence of queries.

We start by considering path languages introduced in Chapter 4. Here all of the languages can be shown to have undecidable containment if the full language is considered, however we do isolate several decidable fragments. These are generally obtained by only allowing queries

to test if two data values are equal and not if they are different. Subclasses defined by such a restriction will be shown to have decidable query containment, with complexity of the problem ranging from PSPACE for *RQD*s to EXPSPACE for *RQM*s and register automata.

Next we investigate the impact of the inverse operator on containment of queries. Remarkably, while adding this operator carries no extra computational cost with respect to query evaluation, it does make a big difference for containment, as now even the subclass that allows only positive data comparisons has undecidable query containment problem.

Having studied path languages we now turn our attention to graph languages. Namely, we consider GXPath and its various dialects. Even though the language was shown to have good computational properties and close connections with logic, when containment is considered the story is quite different: here even the navigational fragment that uses no data value comparisons has undecidable containment problem.

The reason for the undecidability of GXPath is the presence of a powerful negation operator that allows complementation of binary relations. We show, that if one excludes such negation from the language, then containment becomes decidable (EXPTIME-complete). As mentioned before, this language is close to propositional dynamic logic (PDL), whose containment is also known to be EXPTIME-complete [Harel et al., 2000].

Note that so far we only discussed navigational GXPath fragments. In fact, we will mostly concentrate on fragments of $\mathsf{GXPath}_{\mathrm{reg}}$. When data fragments are considered there are still many questions opened and we only present some undecidability results that follow from results about navigational fragments or some of the classes from Chapter 4. The picture is further complicated if we consider core fragments, where most automata theoretic techniques fail [Martens, 2006] and new approaches have to be developed. The situation here is in fact quite similar to the well studied case of XML static analysis where even after several years some of the problems remain unanswered [Benedikt and Koch, 2008, Benedikt et al., 2008], and the ones that have been solved usually require very intricate techniques that cannot be applied in the graph scenario (see e.g. [David et al., 2013, Miklau and Suciu, 2004]).

Overall, we see that when containment is considered, the situation is quite different for languages handling both topology and data than it is for traditional languages allowing only navigational queries. While for the latter containment is generally decidable, we show that for the languages considered here the problem resembles behaviour of relational algebra, where containment is undecidable for the full language, but various restrictions on the use of negation lead to decidable fragments. Hence, the existence of real-world relational systems which deal with similar problems, demonstrates that undecidability or high complexity should not be viewed as an insurmountable obstacle for practical use of the languages studied here, but as a foundation for further research.

To establish the notation we now define the query containment problem formally.

Query containment A query q_1 is *contained* in a query q_2 (written $q_1 \subseteq q_2$) if for each data graph G over Σ and $\mathcal{D}$ we have that every tuple in the answer of q_1 is also in the answer to q_2. The queries q_1 and q_2 are *equivalent* (written $q_1 \equiv q_2$) iff they produce the same answer set for every data graph G.

The containment and equivalence are at the core of many static analysis tasks, such as query optimisation. All the classes of queries considered here are closed under union, so these two problems are easily interreducible: $q_1 \equiv q_2$ iff q_1 and q_2 contains each other, and $q_1 \subseteq q_2$ iff $q_1 \cup q_2 \equiv q_2$. That is why here we concentrate just on the first and consider the following decision problem parametrized by a class of queries $\mathcal{Q}$.

CONTAINMENT ($\mathcal{Q}$)	
Input:	Queries q_1 and q_2 from $\mathcal{Q}$.
Question:	Is q_1 contained in q_2?

Recall that for RPQs query conatinment is equivalent to language containment [Calvanese et al., 2003]. In particular, if we have two RPQs $q_1 = x \xrightarrow{e_1} y$ and $q_2 = x \xrightarrow{e_2} y$, with e_1, e_2 regular expressions, then q_1 is contained in q_2 if and only if the language of e_1 is contained in the language of e_2. From this fact we obtain that containment of RPQs is PSPACE-complete, following the classic result that containment of regular expressions is PSPACE-complete. Since all of the classes of queries studied here are extensions of RPQs, this establishes a lower bound for containment of any of these classes.

Note that for NRQs and NREs defining them, the above claim no longer holds, since they do not define languages, but graph patterns. We will see that path languages and graph languages introduced in Part I and Part II, respectively, exhibit the same behaviour, thus further exemplifying the fundamental differences between them. Note that although we could infer that query containment for graph queries is the same as pattern containment, the containment of patterns is not a standard language theoretic problem, so here we study it in isolation.

Remark 9. *When studying static analysis of a query language that deals with data values it is usual to disregard constants [Segoufin, 2007, Figueira, 2010b] as they often make the presentation more notation heavy. Therefore in the languages considered in this Chapter we will assume that data values are only compared to each other for (in)equality and not compared to constants.*

10.1 Containment of path queries

We begin our study of the query containment by examining the problem for classes of path languages introduced in Part I. Note that throughout this section use graph semantics introduced in Section 5.1, as opposed to the usual path semantics from Chapter 4. This will make some

of the notation less cumbersome, particularly when considering two-way queries. It will also allow us to have a uniform treatment of both one-way and two-way queries, as well as path and graph queries.

It is important to remark that, as discussed in Section 5.1, when using graph semantics we will often abuse the notation and identify the expression defining the query with the query itself. Therefore we will often use e.g. regular expression e to denote both the query $Q = x \xrightarrow{e} y$ and the expression itself. This, however, should cause no confusion as it will always be clear from the context if we are using the query, or the expression defining it.

10.1.1 Containment of RQMs

We start by examining the containment problem for RQM queries. As mentioned in the introduction to this chapter, for path languages query containment is equivalent to language containment. It is readily checked that this holds for RQMs as well.

Lemma 10.1.1. *Given two RQMs $q_1 = x \xrightarrow{e_1} y$ and $q_2 = x \xrightarrow{e_2} y$, where e_1 and e_2 are regular expressions with memory, it holds that $q_1 \subseteq q_2$ iff $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.*

Note that in the proposition above $q_1 \subseteq q_2$ is defined on data graphs, but $\mathcal{L}(e_1)$ and $\mathcal{L}(e_2)$ are sets of data paths.

We now turn to the containment problem for RQMs. Unfortunately, as the following theorem shows, the power that RQMs gain through their data manipulation mechanism comes with a high price for static analysis tasks.

Theorem 10.1.2. *The problem* CONTAINMENT *(*RQMs*) is undecidable.*

This fact follows from Proposition 10.1.1 and the undecidability of the containment problem for regular expressions with memory (Corollary 6.2.7).

The theorem above naturally leads to question of finding decidable subclasses. It is known that testing containment of an expression using at most one register in an expression using at most two registers is decidable [Neven et al., 2004]. This approach appears to be too restrictive, and thus we concentrate instead on *positive RQMs*, i.e. those RQMs, that use only atoms of the form $x^{=}$ in the conditions. In [Tal, 1999] it was shown that the containment of positive RQMs is decidable, but no complexity bounds were given. The following theorem fills this gap.

Theorem 10.1.3. *The problem* CONTAINMENT *(*positive RQMs*) is* EXPSPACE*-complete.*

Proof. To prove the upper bound we will rely on the equivalence of RQMs and register automata. For hardness we do a reduction from acceptance problem of a Turing machine that works in EXPSPACE. We start with the upper bound.

Upper bound. To prove this we will need some auxiliary definitions and claims.

It will be more convenient to show the upper bound for register automata over data paths. Recall that these were defined in Section 4.1.

It was shown in Proposition 4.2.3 that for every RQM e one can construct in polynomial time a register data path automaton $\mathcal{A}$ such that $\mathcal{L}(e) = \mathcal{L}(\mathcal{A})$. Let then e_1 and e_2 be RQMs. To show that $e_1 \subseteq e_2$ we can, by Lemma 10.1.1, show instead that $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$. Moreover, by the aforementioned equivalence with automata, it suffices to show that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ for the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ equivalent to e_1 and e_2.

The reminder of the proof is devoted to showing that such decision problem belongs to EXPSPACE, assuming both $\mathcal{A}_1$ and $\mathcal{A}_2$ use only equalities in the conditions.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two register automata that only use equalities in the conditions, such that $\mathcal{L}(\mathcal{A}_1) \not\subseteq \mathcal{L}(\mathcal{A}_2)$. Then there is a data path $w = d_1 a_1 d_2 a_2 \cdots a_n d_{n+1}$ that belongs to $\mathcal{L}(\mathcal{A}_1)$ but it does not belong to $\mathcal{L}(\mathcal{A}_2)$. Further, there is an accepting run τ that associates to each data value d_i in w a change of configuration, going from a configuration of the form $(2i-1, q, \lambda)$ to one of the form $(2i, q', \lambda')$.

Set $w^1 = w$ and $\tau^1 = \tau$. Starting from $i = 2$ up to $i = n+1$, we repeatedly perform the following operations on w^i, increasing i.

Let w^{i-1} and τ^{i-1} be the resulting data path and accepting run after performing the $i-1$-th operation, and assume that τ_{i-1} changes from a configuration $(2i-1, q, \lambda)$ to $(2i, q', \lambda')$. If all data values in the image of λ are also in the image of λ', then let $w^i = w^{i-1}$ and $\tau^i = \tau^{i-1}$. Otherwise, assume that $d^1, \ldots, d^\ell$ are in the image of λ but not of λ'. Then let $p^1, \ldots, p^\ell$ be fresh, new data values. Construct w^i as follows. For each $j = 1, \ldots, \ell$, replace all appearances of d^j in w_{i-1}, only after position $2i-2$ of w^{i-1}, with the data value p^j. Moreover, construct τ^i by replacing as well $d^1, \ldots, d^\ell$ with $p^1, \ldots, p^\ell$ in all the register values of the remaining configurations, from position $2i-1$ onwards.

For the automaton $\mathcal{A}_1$, data path $w \in \mathcal{L}(\mathcal{A}_1)$ and run τ witnessing the acceptance of w, let us denote by $u_{w,\tau}$ the resulting data path w^{n+1} after performing all transformations above, and by $\sigma_{w,\tau}$ the resulting run τ^{n+1}. Note that the constructed run remains a valid run, so that $\mathcal{A}_1$ accepts as well the path $u_{w,\tau}$. Moreover, the following can be shown about $u_{w,\tau}$ (the proof follows from the construction): if there are positions j_1 and j_2 of $u_{w,\tau}$ such that both j_1 and j_2 contain the same data value, then such data value is present in at least one register in all configurations of $\sigma_{w,\tau}$ starting from position j_1 and ending in position j_2.

Moreover, since the automaton $\mathcal{A}_2$ does not accept w, we have that it does not accept $u_{w,\tau}$. This follows simply because we are only using automata with equalities, and our transformation actually introduce additional inequalities on the data values of paths. From the above facts we obtain the following claim.

Claim 10.1.4. *Given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we have that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ if and only if there is a data path $w \in \mathcal{L}(\mathcal{A}_1)$, accepted by run τ, such that $u_{w,\tau}$ belongs to $\mathcal{L}(\mathcal{A}_1)$ but does not belong to $\mathcal{L}(\mathcal{A}_2)$.*

All that remains now is to show that the existence of such a data path can be decided in EXPSPACE.

Let now $\mathcal{A}_1 = (Q_1, q_1^0, F_1, \lambda_1^0, \delta_1)$ and $\mathcal{A}_2 = (Q_2, q_2^0, F_2, \lambda_2^0, \delta_2)$. Furthermore, assume that REG_1 and REG_2 are all possible assignments of registers in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively (obviously these are infinite sets).

Consider the following transition system. Its states are $Q_1 \times REG_1 \times 2^{Q_2 \times REG_2}$. The initial state is $(q_1^0, \lambda_1^0), \{(q_2^0, \lambda_2^0)\}$, the set of final states are all those states that contain a state in F_1 and do not contain any state in F_2 (i.e. if at any point we are in a final state, we know that a given data path is accepted by $\mathcal{A}_1$ but it is not accepted by $\mathcal{A}_2$).

The transition is defined as follows: there is a transition between state $(q_1, \lambda_1), \{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$ and state $(q_1', \lambda_1'), \{(q'^1_2, \lambda'^1_2), \ldots, (q'^m_2, \lambda'^m_2)\}$ by letter a or data value d if one can go from (q_1, λ_1) to (q_1', λ_1') using δ_1 over a or d, and $\{(q'^1_2, \lambda'^1_2), \ldots, (q'^m_2, \lambda'^m_2)\}$ is the set of all states that are reachable from any state in $\{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$, using δ_2 and a or d.

Now, obviously the size of this transition system is infinite. However, we proceed as follows.

We guess, symbol by symbol, the data path $u_{w,\tau}$ and its run $\sigma_{w,\tau}$, and only pick those moves in the transition system where q_1 and λ_1 move as in $\sigma_{w,\tau}$. Then by the properties of $u_{w,\tau}$ and $\sigma_{w,\tau}$ we know that any state $(q_1, \lambda_1), \{(q_2^1, \lambda_2^1), \ldots, (q_2^n, \lambda_2^n)\}$ can be simplified into a state in which all values in $\lambda_2^1, \ldots, \lambda_2^n$ that are not in λ_1 are mapped to a single fresh value d. This is because such data values will never appear again in $u_{w,\tau}$, and thus from the equality point it is just as good as any data value which is different to all the remaining values in $u_{w,\tau}$.

But we can do even better, as here it suffices to store only the equivalence classes of the registers, i.e. whether the registers store, at any given point, the same data value as in other register, or a different one. If the next symbol we are guessing corresponds to a data value that was in one of the registers of λ_1, then we guess, instead of the particular data value, the following information "the incoming data value is the one stored in register x". The system then updates the equivalence classes according to the registers. If, on the contrary, the incoming data value is a data value different from all λ_1, we just guess "the incoming data value is not stored in any register", and then updates the information as before.

Thus, for our simulation of $\mathcal{A}_1$ it suffices to store, at any given point, the equivalence class formed by the registers in $\mathcal{A}_1$, and to simulate all possible runs of $\mathcal{A}_2$ we need to store, besides the equivalence classes of its registers, a pointer indicating whether it is storing a value also stored in a register of $\mathcal{A}_1$, or whether it is storing a data value not currently stored in $\mathcal{A}_1$ (that

will never show up again in our data path). This amounts to a total of $Q_1 \times 2^{|\mathcal{A}_1|} \times 2^{Q_2 \times 2^{|\mathcal{A}_2| \times |\mathcal{A}_1|}}$ states, which is doubly exponential in $\mathcal{A}_1$ and $\mathcal{A}_2$. We can therefore decide whether there is a valid run fo this system (that ends in a final state) using a standard on-the-fly EXPSPACE algorithm.

Hardness. The proof of EXPSPACE-hardness is by reduction from the complement of the acceptance problem of a Turing machine.

Let L be a language that belongs to EXPSPACE over some alphabet Γ, $\mathcal{M}$ be a deterministic Turing machine that decides L in EXPSPACE, and w be a word (plain, without data values) over Γ. Next we show how to construct RQMs e' and e (in polynomial time in the size of $\mathcal{M}$ and w) such that $\mathcal{L}(e') \subseteq \mathcal{L}(e)$ if and only if $\mathcal{M}$ does not accept the input w. By Proposition 10.1.1 this is enough for the proof of the hardness.

Let $\mathcal{M} = (Q, \Gamma, q_0, \{q_f\}, \delta)$, where $Q = \{q_0, \ldots, q_f\}$ is the set of states, Γ is the tape alphabet, containing the distinguished blank symbol B, q_0 and q_m are the unique initial and final states, and $\delta : (Q \setminus \{q_f\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function. Notice, that without loss of generality we assume that no transition is defined on the unique final state q_f. Since $\mathcal{M}$ decides L in EXPSPACE, there exists a polynomial P (which does not depend on w) such that $\mathcal{M}$ decides w using space 2^n, where $n = P(|w|)$. Let also $w = a_0 a_1 \cdots a_k$.

In what follows we will slightly abuse the notation. Namely, for alphabet $\Delta = \{b_1, \ldots, b_m\}$ of symbols, we denote by the same Δ the regular expression $(b_1 \cup \cdots \cup b_m)$.

Let $\Sigma = \{\#, \&, \%, \triangle\} \cup \Gamma \cup (\Gamma \times Q)$ be the alphabet of the constructing expressions e' and e.

Let $\langle i \rangle$ denote the binary representation of the number i as a data path on n labels # such that its data values represent the string representation of i as a binary number. That is, the data path $d_n \# d_{n-1} \# \ldots \# d_1$ such that $d_n d_{n-1} d_1$ is precisely the string representation of i as a binary number. For example, $\langle 0 \rangle$ is the data path $(0\#)^{n-1}0$, and $\langle 2 \rangle$ is the data path $(0\#)^{n-2}1\#0$.

We represent configurations of the Turing machine by data paths satisfying

$$\langle 0 \rangle (\Gamma \cup (\Gamma \times Q)) d \quad \& \quad \langle 1 \rangle (\Gamma \cup (\Gamma \times Q)) d \quad \& \quad \langle 2 \rangle (\Gamma \cup (\Gamma \times Q)) d \quad \& \quad \ldots$$
$$\langle 2^n - 1 \rangle (\Gamma \cup (\Gamma \times Q)) d \,\&\, d \,\%\, d, \qquad (10.1)$$

where d stands for any data value. Intuitively, the data paths $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 2^n - 1 \rangle$ indicate each of the 2^n cells of $\mathcal{M}$, and the symbol following such a data path represents either the content of the cell (which means that the head does not point here), or the content of the cell plus the state of $\mathcal{M}$ (if $\mathcal{M}$ is pointing at that particular cell at a given point of the computation).

Since every configuration of $\mathcal{M}$ can be represented as a data path of form (10.1), a run of $\mathcal{M}$ on the input w can be seen as a sequence (i.e. concatenation) of data paths of form (10.1).

The idea of the reduction is the following. The expression e' is such that it accepts all data paths in each of which every data value is equal to one of the first two data values of the path. Without loss of generality we can then denote the first data value of each of these data paths by 0 and the second data value by 1. In turn, the expression e shall represent all those data paths that belong to $\mathcal{L}(e')$ that are either not valid concatenations of paths of form (10.1), or that the sequence of configurations is not a valid run of $\mathcal{M}$ on input w (in both cases, followed by some initialisation). This way, if there is a valid run for $\mathcal{M}$ on w, we have that there is a data path in $\mathcal{L}(e')$ that is not in $\mathcal{L}(e)$, i.e. $\mathcal{L}(e') \not\subseteq \mathcal{L}(e)$.

Formally, the first of these expressions e' is defined as following:

$$e' = \downarrow x.\triangle\downarrow y.(\triangle[x^{=}] \cup \triangle[y^{=}])\big(\Sigma[x^{=}] \cup \Sigma[y^{=}]\big)^{*}.$$

We split the definition of the second expression into six parts $e = e^0 \cup e^1 \cup e^2 \cup e^3 \cup e^4 \cup e^5$, such that

- e^0 describes all data paths that use a single data value (instead of two);
- e^1 describes all data paths that are not concatenations of paths of form (10.1);
- e^2 describes all data paths that, even if they are concatenations of paths of form (10.1), some of them do not represent valid configurations for $\mathcal{M}$;
- e^3 describes data paths in which the first configuration does not correctly describe the initial configuration of $\mathcal{M}$ on input w;
- e^4 describes those data paths in which the last sub path of form (10.1) does not represent an accepting configuration of $\mathcal{M}$;
- e^5 describes data paths that contain two consecutive sub paths of form (10.1) that represent configurations for $\mathcal{M}$ which, however, do not agree on δ.

Expression e^0 is straightforward to define. Next we give the remaining ones.

Expression e^1. Most of this expression is not really related to data values, but instead can be defined by an NFA in a standard way (see [Barceló et al., 2013b] Theorem 6). The only interesting part is the one which accepts all data paths with a "configuration" in which "cells" are concatenated not in the only proper order, from $\langle 0\rangle$ to $\langle 2^n - 1\rangle$. To do this we include in e^1 the a disjunction of the following expressions:

- the expressions

  $$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^{*}(\#[x^{=}])^{n}(\Sigma \setminus \{\%\})^{*}(\#[x^{=}])^{n}\Sigma^{*},$$
  $$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^{*}(\#[y^{=}])^{n}(\Sigma \setminus \{\%\})^{*}(\#[y^{=}])^{n}\Sigma^{*},$$

  which look for two data paths of form $\langle 0\rangle$ within one configuration, and likewise for $\langle 2^n - 1\rangle$;

- the expressions

$$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\%\,(\#[x^=])^i\,\#[y^=]\,\Sigma^*, \qquad \text{for each } 0\le i\le n-1,$$
$$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#[x^=]\,(\#[y^=])^i\,(\Gamma\cup(\Gamma\times Q))\,\%\,\Sigma^*, \qquad \text{for each } 0\le i\le n-1,$$

  which look for a configuration starting with something different from $\langle 0\rangle$, and likewise ending with something different from $\langle 2^n-1\rangle$;

- the expression

$$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#^{n-1}\,\#[x^=]\,(\Gamma\cup(\Gamma\times Q))\,\&\,\#^{n-1}\,\#[x^=]\,\Sigma^*,$$

  looking for a configuration where an even number follows with another even number;

- the expressions

$$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#^i\,\#[x^=]\,\#^{n-i-2}\,\#[x^=]\,(\Gamma\cup(\Gamma\times Q))\,\&\,\#^i\,\#[y^=]\,\#^{n-i-1}\,\Sigma^*, \quad 0\le i\le n-2,$$
$$\downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\,\#^i\,\#[y^=]\,\#^{n-i-2}\,\#[x^=]\,(\Gamma\cup(\Gamma\times Q))\,\&\,\#^i\,\#[x^=]\,\#^{n-i-1}\,\Sigma^*, \quad 0\le i\le n-2,$$

  looking for a configuration where an even number follows with a number where some of the digits are different from the onces in the previous number (except the last).

Note that last 2 cases cover all configurations in which even position numbers are not followed by their successors. It is also possible, but rather cumbersome and lengthy, to define expressions which cover the even—odd cases. We omit such definition, and refer the reader to [Barceló et al., 2013b] for very similar constructions.

Expression e^2. Similarly to the next expressions e^3 and e^4, it can be described with standard NFA's. In particular, e^2 is the union of expressions stating the following:

- between two symbols % there is no symbol in $(\Gamma\times Q)$, which means that in some configuration the machine dos not point to any cell;

- between two neighbouring symbols % there are two symbols in $(\Gamma\times Q)$, which means that the machine is pointing at two cells.

Expression e^3. It is the union of expressions stating the following:

- the first configuration does not contain the initial state in the first position of the tape, reading the first symbol of the input;

- the following $k-1$ cells do not contain the remainder of the input;

- any of the remaining cells does not contain the blank symbol.

Expression e^4. It can be dfined in the similar way as e^3.

Expression e^5. It is defined as the union of the following expressions:

- a cell not pointed by the head changed its content from one configuration to the subsequent one:

$$\bigcup_{a\in\Gamma} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\#\downarrow x_1.\#s\downarrow x_{n-1}.\#\downarrow x_n.a\,(\Sigma\setminus\{\%\})^*\,\%\\ (\Sigma\setminus\{\%\})^*\#[x_1^=]\#[x_2^=]s\#[x_n^=]\left((\Gamma\setminus\{a\})\cup((\Gamma\setminus\{a\})\times Q)\right)\Sigma^*;$$

- a configuration which is not final features a pair in $\Gamma\times Q$ for which no transition is defined

$$\bigcup_{\{(a,q)|\delta(q,a)\text{ is not defined}\}} \Sigma^*\,(a,q)\,\Sigma^*\,\%\,\Sigma^+;$$

- the change of state does not agree with δ:

$$\bigcup_{\{(a,q)|\delta(q,a)=(a',q',\{L,R\})\}} \Sigma^*\,(a,q)\,(\Sigma\setminus\{\%\})^*\,\%\,(\Sigma\setminus\{\%\})^*\left(\Gamma\times(Q\setminus\{q'\})\right)\Sigma^*;$$

- the symbol written in a given step does not agree with δ:

$$\bigcup_{\{(a,q)|\delta(q,a)=(a',q',\{L,R\})\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\#\downarrow x_1.\#s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%\\ (\Sigma\setminus\{\%\})^*\#[x_1^=]\#[x_2^=]s\#[x_n^=]\,(\Gamma\setminus\{a'\})\,\Sigma^*;$$

- the movement of the head does not agree with δ:

$$\bigcup_{\{(a,q)|\delta(q,a)=(a',q',R)\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\#\downarrow x_1.\#s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%\\ (\Sigma\setminus\{\%\})^*\#[x_1^=]\#[x_2^=]s\#[x_n^=]a'\,\&\,(\varepsilon\cup\left(\#^n\,\Gamma\,(\Sigma\setminus\{\%\})^*\right))\,\%\,\Sigma^*,$$

$$\bigcup_{\{(a,q)|\delta(q,a)=(a',q',L)\}} \downarrow x.\triangle\downarrow y.\triangle\,\Sigma^*\#\downarrow x_1.\#s\downarrow x_{n-1}.\#\downarrow x_n.(a,q)\,(\Sigma\setminus\{\%\})^*\,\%\\ (\varepsilon\cup\left((\Sigma\setminus\{\%\})^*\#^n\,\Gamma\,\&\right))\#[x_1^=]\#[x_2^=]s\#[x_n^=]\,\Sigma^*.$$

With these definitions in hand, it is now straightforward to show that $\mathcal{L}(e')\subseteq\mathcal{L}(e)$ if and only if $\mathcal{M}$ does not accept on input w. This finishes the proof of the EXPSPACE lower bound.

□

The previous proof relies on the fact that the set of variables used in our queries is unbounded. Carefully checking the proof reveals the following corollary. Here *n-bounded positive RQMs* refers to the class of positive RQMs which can use at most n variables (that is they are defined using conditions from $\mathcal{C}_k$, for a fixed k).

Corollary 10.1.5. *Let n be a natural number. The problem* CONTAINMENT *(*n-bounded positive RQMs*) is* PSPACE*-complete.*

Hence, positive RQMs are a natural subclass of RQMs with decidable query containment. However, when comparing the complexity with the one for RPQs, we see that allowing positive data test comparisons results in an exponential jump. In the following section we will see that positive RQDs form a class of queries with complexity of the containment problem matching that of RPQs.

10.1.2 Containment of RQDs

Similarly as for RQMs, we can show an analogue of Proposition 10.1.1, thus reducing query containment to language containment.

Proposition 10.1.6. *Given two RQDs* $q_1 = x \xrightarrow{e_1} y$ *and* $q_2 = x \xrightarrow{e_2} y$*, it holds that* $q_1 \subseteq q_2$ *iff* $\mathcal{L}(e_1) \subseteq \mathcal{L}(e_2)$.

RQDs were originally introduced as a restriction of RQMs that enjoys much better query evaluation properties. In light of this result, one might also hope for good behaviour when query containment is considered. Surprisingly, the following theorem shows that this is not the case.

Theorem 10.1.7. *The problem* CONTAINMENT *(*RQDs*) is undecidable.*

Proof. We will in fact prove a stronger result stating that the universality problem for regular expressions with equality , defined below, is undecidable. Let $\Sigma[\mathcal{D}]^*$ denote the set of all data paths over the alphabet Σ and set of data values $\mathcal{D}$.

UNIVERSALITY OF REWES	
Input:	A REWE e.
Qestion:	Does $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$?

The undecidability of this problem immediately implies that given regular expressions with equality e_1 and e_2, checking whether $L(e_1) \subseteq L(e_2)$ is undecidable. The latter then implies undecidability of query containment over graphs by Proposition 10.1.6.

The proof of undecidability of universality problem for RQDs is similar to the proof of the universality of register automata in [Neven et al., 2004]. The reduction is from *Post correspondence problem (PCP)*, which is well-known to be undecidable.

An *instance of PCP* is a set of pairs of words

$$\{(u_1, v_1), \ldots, (u_n, v_n)\}, \tag{10.2}$$

over a finite alphabet Γ. A *solution* for an instance I is a sequence $k_1, \ldots, k_m$ of numbers from $\{1, \ldots, n\}$ such that $u_{k_1} \cdots u_{k_m} = v_{k_1} \cdots v_{k_m}$. The question is whether an instance has a solution.

Throughout the reduction we will use the following notation for every data path $w = d_1 a_1 d_2 \ldots a_{k-1} d_k$. Let $\mathrm{REV}(w)$ be the reversal of w, that is $\mathrm{REV}(w) = d_k a_{k-1} \ldots d_2 a_1 d_1$. Also, let $\mathsf{Proj}(w)$ be its projection to the labels, i.e. the word $a_1 \ldots a_{k-1}$.

Let $\$, \#$ be two special symbols not in Γ, let $\Sigma' = \Gamma \cup \{\$, \#\}$, and let $\Sigma = \Gamma \cup \{\$\}$. A solution $k_1, \ldots, k_m$ of a PCP instance I of the form (10.2) can be encoded as a data path $w_1 \# \mathrm{REV}(w_2)$ over Σ, where

$$w_1 = \; 0 \;\; \$c_1 \; a_1 d_1 \cdots a_{\ell_1} d_{\ell_1} \;\; \$c_2 \; a_{\ell_1+1} d_{\ell_1+1} \cdots a_{\ell_1+\ell_2} d_{\ell_1+\ell_2} \cdots\cdots \\ \$c_m \; a_{\ell_1+\cdots+\ell_{m-1}+1} d_{\ell_1+\cdots+\ell_{m-1}+1} \cdots a_{\ell_1+\cdots+\ell_m} d_{\ell_1+\cdots+\ell_m},$$

$$w_2 = \; 0 \;\; \$g_1 \; b_1 f_1 \cdots b_{\ell_1} f_{\ell_1} \;\; \$g_2 \; b_{\ell_1+1} f_{\ell_1+1} \cdots b_{\ell_1+\ell_2} f_{\ell_1+\ell_2} \;\cdots\cdots \\ \$g_m \; b_{\ell_1+\cdots+\ell_{m-1}+1} f_{\ell_1+\cdots+\ell_{m-1}+1} \cdots b_{\ell_1+\cdots+\ell_m} f_{\ell_1+\cdots+\ell_m},$$

such that a's and b's are labels from Σ, c's, g's, d's, f's, and 0 are data values, and, for a shortcut $\ell = \ell_1 + \cdots + \ell_m$, the following conditions hold:

(C1) the symbol # appears only once;

(C2) $\mathsf{Proj}(w_1) \in (\$u_1 \cup \cdots \cup \$u_n)^*$;

(C3) $\mathsf{Proj}(w_2) \in (\$v_1 \cup \cdots \cup \$v_n)^*$;

(C4) the data values c_i's and d_i's are pairwise different;

(C5) the data values g_i's and f_i's are pairwise different;

(C6) $c_1 = g_1$ and $c_m = g_m$;

(C7) $d_1 = f_1$ and $d_\ell = f_\ell$;

(C8) for each $i, j \in \{1, \ldots, m-1\}$ if $c_i = g_j$ then $c_{i+1} = g_{j+1}$;

(C9) for each $i, j \in \{1, \ldots, \ell-1\}$, if $d_i = f_j$ then $d_{i+1} = f_{j+1}$;

(C10) for each $i, j \in \{1, \ldots, \ell\}$, if $d_i = f_j$, then $a_i = b_j$;

(C11) for each $i, j \in \{1, \ldots, m\}$, if $c_i = g_j$, then
$(a_{\ell_1+\ldots+\ell_{i-1}+1} \cdots a_{\ell_1+\ldots+\ell_i}, b_{\ell_1+\ldots+\ell_{j-1}+1} \cdots b_{\ell_1+\ldots+\ell_j}) \in I$.

Note that e.g. Conditions (C4–C6, C8) forces the sequence of c's in w_1 to be equal to the sequence of g's in w_2.

It is straightforward to show that there exists a solution to the PCP instance I if and only if there exists a data path of the form $w_1\#\mathrm{REV}(w_2)$ over Σ' that satisfies Conditions (C1–C11) above. Data path w_1 is meant to encode the u-part of I and w_2 the v-part. The idea is that the equality $c_i = g_i$ codes a position k_i in a solution by a unique data value, and in (C11) it is checked that the pair on this position belongs to I. Also, d's and f's code the actual pairs (u_i, v_i) in I and since we check that d's equal f's in Conditions (C4–C9) and that the letter after each d equals the corresponding one before the appropriate f in Condition (C10). Note that we require data path w_2 to be reversed in order to nest equality tests according to the semantics of REWEs.

We now construct a REWE e over Σ' that accepts a data path w such that it is either not of the form $w_1\#\mathrm{REV}(w_2)$, or at least one of the Conditions (C1–C11) above is not satisfied. Thus, if e is universal (i.e. accepts all data paths) then in particular there is no data path coding a solution to the PCP instance, and, hence there is no solution by itself. The REWE e is obtained by taking the union of the following, using the usual shortcut Δ for the expression $b_1 + \ldots + b_p$ over any alphabet $\Delta = \{b_1, \ldots, b_p\}$:

- REWEs recognising the negations of Conditions (C1–C3), which can be written as standard regular expressions without equality tests;

- the REWE
$$\Big(\Sigma^*\$(\Gamma\Sigma^*\$)_{=}\Sigma^* \quad \cup \quad \Sigma^*\$\Gamma\Gamma^*(\Sigma^*\Gamma)_{=}\Big)\#\Sigma^*,$$
which recognises the negation of (C4); here the left part of $\cup$ finds equal c's, while the right one finds equal d's; note that for equal ds we take care that we don't incidentally compare with some c;

- a REWE which recognises the negation of (C5), which is very similar to the previous one, but takes into account that w_2 is reversed;

- the REWE
$$\$(\Sigma^*)_{\neq}\$ \quad \cup \quad \Sigma^*\$(\Gamma^*\#\Gamma^*)_{\neq}\$\Sigma^*,$$
which recognises the negation of (C6); note, that here we use the fact that w_2 is reversed, so in particular g_1 appears as the second last data value (and right before the final \$), which is covered by the left disjunct; similarly c_m is the value after the last \$ in w_1, so after that we can only advance by means of Γ before reaching # and then we proceed in w_2 to the first \$ in front of which g_m is located;

- a REWE which recognises the negation of (C7), which is very similar to the previous one;

- the REWE

$$\Sigma^*\$(\Gamma^*\$(\Sigma^*\#\Sigma^*\$)_{\neq}\Gamma^*\$)_{=}\Sigma^*,$$

  which recognises the negation of (C8);

- REWEs which recognise the negation of (C9–11), which are very similar to the previous one.

It is straightforward to see that the PCP instance I has no solution if and only if $\mathcal{L}(e) = \Sigma[\mathcal{D}]^*$. This concludes our proof of Theorem 10.1.7. □

This naturally opens the search for subclasses of RQDs with decidable containment problem. Similarly to positive RQMs, one can consider the class of *positive RQDs*, i.e. RQDs where subexpressions of the form $e_{\neq}$ are not allowed. Note that if we apply the procedure described in Proposition 4.4.2 to a positive RQD we end up with a positive RQM. Hence, we again have a strict containment of the corresponding classes, and from Theorem 10.1.3 we conclude that containment of positive RQDs is decidable and in EXPSPACE. However, it was shown in [Kostylev et al., 2014] that we can perform even better, in fact, the best possible in light of the PSPACE lower bound for plain RPQs.

Theorem 10.1.8 ([Kostylev et al., 2014])**.** *The problem* CONTAINMENT *(*positive RQDs*) is* PSPACE*-complete.*

Using the results about containment of RQDs and RQMs we can also deduce the following about RQBs.

Corollary 10.1.9. *Query containment is undecidable for the class of RQB queries. It becomes decidable if we disallow testing for inequalities in conditions.*

Here undecidability follows from Theorem 10.1.7 and the fact that RQBs subsume RQDs. That the positive fragent is decidable is a consequence of Theorem 10.1.3.

10.1.3 Impact of inverse on containment

The classic result by Calvanese et al. [Calvanese et al., 2003] states that one can add the inverse operator to RPQs and maintain not only the same complexity of query evaluation, but also the same complexity of query containment. Since adding inverses to RQMs and RQDs does not affect the complexity of query evaluation this gives a hope that it will also not affect the complexity of containment of 2RQMs and 2RQDs. Of course, by the results of previous sections, containment is undecidable when full languages are considered. Unfortunately, as we show next, decidability for positive RQMs does not propagate to their two-way variant.

The class of *positive* 2RQMs is defined as the subclass of 2RQMs that use only conditions built from atoms of the form $x^{=}$ (but not $x^{\neq}$). Note that for 2RQMs we can no longer use language containment to check for query containment [Calvanese et al., 2003]. Indeed, it might be tempting to do the same as we did for Proposition 5.1.3, and reduce containment checking of two-way queries to containment of the same queries, but viewed as one-way queries over the extended alphabet containing symbols a^- for each $a \in \Sigma$. However, this does not imply that queries are contained, because labels of the form a^- can also symbolise going backwards (for example, query a is contained in aa^-a, but they are not contained when viewed as regular expressions over the extended alphabet). This leads to the following result.

Theorem 10.1.10. *The problem* CONTAINMENT *(*positive 2RQMs*) is undecidable.*

Proof. The proof is by reduction from the problem of non-emptiness of deterministic, stateless 2-way 3-head automata, which was shown to be undecidable in [Yang et al., 2008].

Formally, a *deterministic stateless 2-way 3-head automaton* (or, *DS23A*) over a finite alphabet Γ is given by a transition partial function $\delta : \Sigma \times \Sigma \times \Sigma \rightharpoonup \{-1,0,1\}^3$, where $\Sigma = \Gamma \cup \{\vdash, \dashv\}$, the latter symbols assumed not to be in Γ. These automata accept language of words of form $\vdash \sigma \dashv$, with σ a word over Γ. The automaton starts with its 3 heads reading the $\vdash$ symbol of just before σ, moves its heads according to δ (-1 denotes "move one cell back", 0—"no move", and 1—"move one cell forward"), and accepts σ if at any step of computation over this word all 3 heads point at the symbol $\dashv$.

Let $\mathcal{A}$ be a DS23A. We now construct 2RQMs e' and e over Σ such that the language of $\mathcal{A}$ is empty if and only if $e' \subseteq e$.

The definition of e' is as follows:

$$e' \quad = \quad \vdash \Gamma^* \dashv .$$

As expected, the definition of e is much more intricate. But before it we present a crucial claim.

Claim 10.1.11. *Let e' be the RPQ defined as above, and let e be a 2RQM. Then $e' \subsetneq e$ is and only if there exists a graph G_w corresponding to a data path w with start and end nodes u and v (see Figure 5.2), respectively, such that $(u,v) \in [\![e']\!]^{G_w}$ but $(u,v) \notin [\![e]\!]^{G_w}$.*

Proof. The if direction is obvious, so we only show the only if direction. Assume then that $e' \subsetneq e$. Then there is a graph G and a pair (u',v') of nodes in G such that $(u',v') \in [\![e']\!]^G$ but $(u',v') \notin [\![e]\!]^G$. Consider a data path w which is a projection of labels and data values of a path in G witnessing e'. Then let us consider the graph G_w corresponding to w, with start and end nodes u and v, respectively. Clearly, $(u,v) \in [\![e']\!]^{G_w}$. Now assume for the sake of contradiction that $(u,v) \in [\![e]\!]^{G_w}$. By examining the definition of 2RQMs one immediately

obtains that $(u,v) \in [\![e]\!]^G$, which results in a contradiction. This implies that $(u,v) \notin [\![e]\!]^G$, which was to be shown. □

Next we continue with the definition of e. The idea is the following. Since $\mathcal{A}$ is deterministic, if $\mathcal{A}$ accepts some word σ then there exists a single run that leads to this acceptance. We can take advantage of this determinism, and code with e all computations of $\mathcal{A}$ that end up failing at some point. This way, if there is a data path with a corresponding data graph accepted by e', which is not accepted by e, then the language of $\mathcal{A}$ is nonempty, as $\mathcal{A}$ really accepts this word.

The definition of e is split into three parts as follows:

$$e \quad = \quad e_{\text{eq}} \cup e_{\text{crash}} \cup e_{\text{notdef}}.$$

Intuitively, e_{eq} accepts all graphs corresponding to data paths that have two equal data values (data values shall be used as placeholders for the positions of the heads of $\mathcal{A}$, as will be explained shortly); e_{crash} corresponds to data paths for which the computation of $\mathcal{A}$ crashes, and e_{notdef} corresponds to all data paths for which the computation of $\mathcal{A}$ ends up in a position that is not defined.

The part e_{eq} is straightforward to define. For definitions of the other parts of e we first need to describe the 2RQM e_{valid}, that simulates the computation of $\mathcal{A}$ on its input.

For each (a,b,c) in Σ^3 for which δ is defined, assume that $\delta(a,b,c) = (t_1,t_2,t_3)$, where each t_i is either -1, 0 or 1. Then let $e_{(a,b,c)}$ be the following expression:

$$(\Sigma^-)^* \vdash \Sigma^*[x_1^=]\, a\, (\Sigma^-)^* \vdash \Sigma^*[x_2^=]\, b\, (\Sigma^-)^* \vdash \Sigma^*[x_3^=]\, c$$
$$(\Sigma^-)^* \vdash \Sigma^*[x_1^=]\, r_1\, (\Sigma^-)^* \vdash \Sigma^*[x_2^=]\, r_2\, (\Sigma^-)^* \vdash \Sigma^*[x_3^=]\, r_3,$$

where, as usual, Σ stands for the union of all symbols in the alphabet Σ, Σ^- stands for the union of inverses of all symbols in Σ, and for each i, $1 \leq i \leq 3$,

$$r_i = \begin{cases} \Sigma^- \downarrow x_i. , & \text{if } t_i = -1, \\ \varepsilon, & \text{if } t_i = 0, \\ \Sigma \downarrow x_i. , & \text{if } t_i = 1. \end{cases}$$

Having this construction in hands, let

$$e_{\text{valid}} \quad = \quad \# \downarrow x_1. \downarrow x_2. \downarrow x_3. \left(\bigcup_{(a,b,c) \text{ s.t. } \delta(a,b,c) \text{ is defined}} e_{(a,b,c)} \right)^* .$$

This expression, so far, describes valid computations, up to some step. In order to make sure that we represent all words not accepted by $\mathcal{A}$, we need to accept all words in which this

route of valid computation leads to either a crash (by moving out of the word), or to a transition that is not defined.

Specifically, to describe that a run goes out from the computation space, we define

$$e_{\text{crash}} \quad = \quad e_{\text{valid}} \left(\bigcup_{i=1,2,3} \left(\left((\Sigma^-)^* [x_i^=] \vdash \right) \cup \left(\Sigma^* [x_i^=] \dashv^- \right) \right) \right).$$

Furthermore, for each (a,b,c) such that $\delta(a,b,c)$ is not defined, except $(\dashv,\dashv,\dashv)$ (because this is the final step of an accepting computation), define

$$e_{\neg(a,b,c)} \quad = \quad (\Sigma^-)^* \vdash \Sigma^* [x_1^=]\, a\, (\Sigma^-)^* \vdash \Sigma^* [x_2^=]\, b\, (\Sigma^-)^* \vdash \Sigma^* [x_3^=]\, c,$$

and then

$$e_{\text{notdef}} \quad = \quad e_{\text{valid}} \left(\bigcup_{(a,b,c) \text{ s.t. } \delta(a,b,c) \text{ is not defined, and } (a,b,c) \neq (\dashv,\dashv,\dashv)} e_{\neg(a,b,c)} \right).$$

It is now straightforward to show that the language of $\mathcal{A}$ is nonempty if and only if there exists a graph G_w corresponding to a data path w with start and end nodes u and v, respectively, such that $(u,v) \in [\![e']\!]^{G_w}$ but $(u,v) \notin [\![e]\!]^{G_w}$. Application of Claim 10.1.11 finishes the proof of the theorem. □

This negative result comes as a surprise, and it poses a question on whether the containment problem is at least decidable for positive 2RQDs. We leave this question for future work.

10.1.4 Containment of Variable automata

It is known that the language containment problem for VFAs is undecidable [Grumberg et al., 2010a]. Since query containment for RQVs is equivalent to language containment of underlying VFAs it readily follows that the problem of checking, for two RQVs Q_1, Q_2 if $Q_1(G) \subseteq Q_2(G)$, for every data graph G, is undecidable too. Thus we get that:

Proposition 10.1.12. *The problem* CONTAINMENT *(*RQVs*) is undecidable.*

As mentioned previously to obtain decidable language containment one has to restrict to deterministic VFAs (see Fact 6.5.7). These then give a rise to a subclass of *RQV*s with decidable query containment.

Proposition 10.1.13. *The containment problem for queries posted by* deterministic *VFAs is in* CONP.

10.2 GXPath and its many fragments

In this section we study the containment problem for various fragments of GXPath. As mentioned previously, here we can no longer reduce containment over graphs to containment over data paths as we did for RQMs and RQDs in Lemmas 10.1.1 and 10.1.6. To see this consider e.g. GXPath query $a[b^+]c$. This query will select all nodes connected by a path labelled ac, with the intermediate node having an arbitrary sequence of outgoing b-labelled edges. The pattern described by this query is illustrated in the following image.

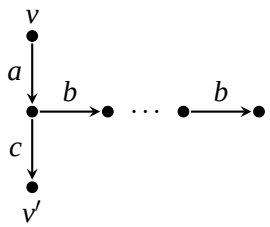


Figure 10.1: A pattern for GXPath query $a[\langle b^+ \rangle]c$.

It is straightforward to see that such a query is not satisfiable on words, while it is on graphs. From this it readily follows that containment over graphs differs from containment over words.

We begin our study by considering navigational fragments of $\mathsf{GXPath}_{\mathrm{reg}}$ first, moving to extensions allowing data value tests later on.

10.2.1 Containment of navigational languages

Analysing the expressive power of $\mathsf{GXPath}_{\mathrm{reg}}$ reveals that this class of queries is equivalent to the extension of first order logic with three variables (FO^3) with the transitive closure operator (see Theorem 7.3.5). It is well known that satisfiability of FO^3 formulas is undecidable over arbitrary (possibly infinite) graphs, and it is folklore to assume that this bound is maintained for finite graphs studied here. Since containment is a more general problem than satisfiability we immediately obtain undecidability for $\mathsf{GXPath}_{\mathrm{reg}}$. As we could not find a formal proof of the aforementioned result about finite satisfiability of FO^3 in the literature, we include a self contained proof below.

Theorem 10.2.1. *The* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}$*) problem is undecidable.*

The proof shows that even satisfiability problem for $\mathsf{GXPath}_{\mathrm{reg}}$ formulas is undecidable. To obtain this result we give a reduction from a variation of tiling problem from [Gurevich and Koryakov, 1972]. In particular we use the fact that the set $\mathcal{S}_{\mathrm{notiling}}$, of all finite sets of tiles that can not tile the positive plane, and the set $\mathcal{S}_{\mathrm{period}}$, of all finite sets of tiles that can tile the plane periodically, are recursively inseparable.

Following the ideas from [Goldblatt and Jackson, 2012], we then show how to construct, for each finite set of tiles $\mathcal{T}$, a $\mathsf{GXPath}_{\text{reg}}$ node formula $\gamma_{\mathcal{T}}$ such that satisfiability of $\gamma_{\mathcal{T}}$ implies that $\mathcal{T}$ can tile the positive plane, while the fact that $\mathcal{T}$ can tile the plane periodically implies that $\gamma_{\mathcal{T}}$ is satisfiable. Note that this shows that the set $S = \{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\text{period}}\}$ and is disjoint from $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\text{notiling}}\}$. The fact that $\mathcal{S}_{\text{notiling}}$ and $\mathcal{S}_{\text{period}}$ are recursively inseparable then implies that S can not be recursive, so satisfiability, and thus containment, of $\mathsf{GXPath}_{\text{reg}}$ queries is undecidable.

To define the formula $\gamma_{\mathcal{T}}$ we rely heavily on the fact that $\mathsf{GXPath}_{\text{reg}}$ can force loops in a graph, thus allowing us to check that tiles are placed correctly and that the tiling can proceed from any point in the plane.

We now give the full proof.

Proof. The proof follows the main lines of the proof of undecidability of PDL with extras from [Goldblatt and Jackson, 2012]. To deduce undecidability we do a reduction from a variant of the tiling problem shown to be undecidable in [Gurevich and Koryakov, 1972] and [Börger et al., 1997].

First we define the terminology needed to state the problem precisely.

A *finite set of tiles* is a collection $\mathcal{T} = \{T_1, \ldots, T_k\}$ of *square tiles*, together with two *edge* relations $\sim_h$ and $\sim_v$. The fact that $T_i \sim_h T_j$ means that the tile T_j can be placed to the right of the tile T_i in a horizontal row, while $T_i \sim_v T_j$ means that T_i can be placed below T_j in a vertical column.

A tiling of the non-negative grid $\mathbb{N} \times \mathbb{N}$ is a function from $t : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ such that for all i, j

- $t(i, j) \sim_h t(i+1, j)$ and,
- $t(i, j) \sim_v t(i, j+1)$.

Tilings of integer grid $\mathbb{Z} \times \mathbb{Z}$ are defined analogously. We say that a set of tiles can tile $\mathbb{Z} \times \mathbb{Z}$ periodically if there is a tiling of $\mathbb{Z}_n \times \mathbb{Z}_m$ for some positive integers n and m that can be used to tile the entire grid by repeating this segment both vertically and horizontally. One can imagine this tiling as forming a torus since the bottom row can be "glued" to the top one and the same for left and right edge of this finite grid.

Let now S_{notiling} denote the set of all finite sets of tiles that can *not* tile $\mathbb{N} \times \mathbb{N}$ and let S_{period} be the set of all finite sets of tiles that can tile $\mathbb{Z} \times \mathbb{Z}$ periodically.

To prove undecideability we will use the following fact.

Fact 10.2.2. ([Gurevich and Koryakov, 1972, Börger et al., 1997]) *Sets S_{notiling} and S_{period} are recursively inseparable. In particular there is no recursive set S such that $S_{\text{period}} \subseteq S$ and $S_{\text{notiling}} \cap S = \emptyset$.*

Fix the finite alphabet of edge labels $\Sigma = \{U, D, L, R, a\}$. In what follows U is meant to interpret "up", D "down", L "left" and R "right", while a will be used to code the tiles. Note that we can work with only $\{U, R, a\}$, since we can use U^- instead of D ans R^- instead of L, but we opted for the extended alphabet to make the formulas easier to understand.

Let now $\mathcal{T} = \{T_1, \ldots, T_k\}$ be a finite set of tiles. For $i = 1 \ldots k$ define $\alpha_i = \langle a^i \cap \varepsilon \rangle$. In what follows α_i is meant to denote the placement of the tile T_i at some position in the grid. E.g. $\langle aaa \cap \varepsilon \rangle$ will denote the placement of the tile T_3 and so on.

We also define the following node formulas of GXPath that will be used throughout the proof. First, for every path formula β we define

$$\mathtt{loop}(\beta) := \langle \beta \cap \varepsilon \rangle \wedge \neg \langle \beta \cap \overline{\varepsilon} \rangle.$$

This formula extracts all nodes v from the graph that have an outgoing β path and such that every such path ends at v itself. It is easy to check that for any graph database G:

$$[\![\mathtt{loop}(\beta)]\!]^G = \{v \in G \mid (\exists v') \text{ s.t. } (v, v') \in [\![\beta]\!]^G \text{ and } (\forall v') \text{ if } (v, v') \in [\![\beta]\!]^G \text{ then } v = v'\}.$$

Second, for every path expression β and every node test φ we define the following formula:

$$\mathtt{when}(\beta, \varphi) := \neg \langle \beta [\neg \varphi] \rangle.$$

The intended meaning of this node formula is to extract all nodes v from a graph such after every β-path starting in v ends with a node belonging to $[\![\varphi]\!]^G$. Again, it is easy to check that for any graph database G:

$$[\![\mathtt{when}(\beta, \varphi)]\!]^G = \{v \in G \mid (\forall v') \text{ if } (v, v') \in [\![\beta]\!]^G \text{ then } v' \in [\![\varphi]\!]^G\}.$$

Associated with the set of tiles $\mathcal{T}$ we define the formula $\gamma_{\mathcal{T}} = \gamma_1 \wedge \gamma_2$.

To define our formula γ_1 we need to be able to force a "square" at any position in our model, both in a clockwise and in anticlockwise direction. This is done by the means of formula `square` which is defined as the conjunction of the following two formulas:

$$\begin{aligned} \mathtt{clockwise} := \; & \mathtt{loop}(U \cdot D) \wedge \mathtt{when}(U, \mathtt{loop}(R \cdot L)) \wedge \mathtt{when}(U \cdot R, \mathtt{loop}(D \cdot U)) \\ & \wedge \mathtt{when}(U \cdot R \cdot D, \mathtt{loop}(L \cdot R)) \wedge \mathtt{loop}(U \cdot R \cdot D \cdot L), \end{aligned}$$

$$\begin{aligned} \mathtt{anticlockwise} := \; & \mathtt{loop}(R \cdot L) \wedge \mathtt{when}(R, \mathtt{loop}(U \cdot D)) \wedge \mathtt{when}(R \cdot U, \mathtt{loop}(L \cdot R)) \\ & \wedge \mathtt{when}(R \cdot U \cdot L, \mathtt{loop}(D \cdot U)) \wedge \mathtt{loop}(R \cdot U \cdot L \cdot D). \end{aligned}$$

Intuitively `clockwise` allows us to define a square starting at some point in our graph and going "up", then "right", then "down" and finally "left", finishing at the same point. It also forces the point to be able to complete the square whenever it has an outgoing "up" arrow U.

Similarly anticlockwise forces a square starting with "right" and completing it in an obvious way.

Now γ_1 simply states that we can make a square at any point:

$$\gamma_1 := \mathtt{when}(U^*, \mathtt{when}(R^*, \mathtt{square})).$$

Formula γ_2 is going to be responsible for forcing a tiling and is defined next. First, let

$$\alpha = \bigvee_{i=1\ldots k} \alpha_i \wedge \bigwedge_{i=1\ldots k} (\alpha_i \to \bigwedge_{j\neq i} \neg\alpha_j).$$

Note that α simply states that precisely one α_i is true. Here and in the remainder of the proof we use the node formula $\varphi \to \psi$ as a shorthand for $\neg\varphi \vee \psi$.

Next for each i, define β_i as the disjunction of all the α_j such that $T_i \sim_h T_j$. That is β_i is a disjunction of all the tiles that can be placed to the right of the tile i. Similarly, define β^i to be the disjunction of all α_j such that $T_i \sim_v T_j$.

Now let tile be the formula denoting that a tile is placed correctly in the grid. Formally:

$$\mathtt{tile} := \alpha \wedge \bigwedge_{i=1\ldots k} (\alpha_i \to (\mathtt{when}(R, \beta_i) \wedge \mathtt{when}(U, \beta^j))).$$

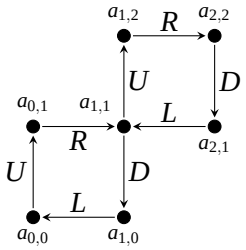
Finally define

$$\gamma_2 := \mathtt{when}(U^*, \mathtt{when}(R^*, \mathtt{tile})).$$

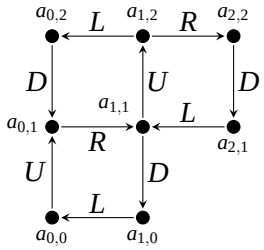
We now show how to deduce the wanted reduction. More formally we show that the set $\{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\text{period}}\}$ and is disjoint from $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\text{notiling}}\}$. Note that Fact 10.2.2 implies that $\{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ can not be recursive.

First we show that if $[\![\gamma_{\mathcal{T}}]\!]^G \neq \emptyset$ for some graph G, then $\mathcal{T}$ can tile the positive plane $\mathbb{N} \times \mathbb{N}$. Take any node $a_{0,0} \in [\![\gamma_{\mathcal{T}}]\!]^G$. By γ_1 the proposition square has to be true at $a_{0,0}$, so in particular $\mathtt{loop}(U \cdot D)$ is true. This means that there is a point which we label $a_{0,1}$ that can be reached from $a_{0,0}$ by an U-labelled edge. (Note that we can also get from $a_{0,1}$ to $a_{0,0}$ by and D-labelled edge.) Now since $\mathtt{when}(U, \mathtt{loop}(R \cdot L))$ is also true at $a_{0,0}$, there must be a node which we label $a_{1,1}$, reached by an R-labelled edge from $a_{0,1}$ (and with the corresponding L-labelled edge in the other direction). Again, this time using the fact that $\mathtt{when}(U \cdot R, \mathtt{loop}(D \cdot U))$ is true at $a_{0,0}$, we get a node labelled $a_{1,0}$, connected to $a_{1,1}$ by an D-labelled edge (and with an U-labelled edge connecting it back with $a_{1,1}$). Next, we use the fact that $\mathtt{when}(U \cdot R \cdot D, \mathtt{loop}(L \cdot R))$ is true at $a_{0,0}$ to get a node $a'_{0,0}$ to the left of $a_{1,0}$. Finally, since $\mathtt{loop}(U \cdot R \cdot D \cdot L)$ is true at $a_{0,0}$, it must be that $a'_{0,0} = a_{0,0}$. Again we note that each edge has a dual edge with the appropriate label, connecting the node in reverse direction.

Similarly, since square is true at $a_{1,1}$ (as we can reach it from $a_{0,0}$ by traversing U and then R-labelled edge), we can also find points $a_{1,2}, a_{2,2}$ and $a_{2,1}$ in an analogous way. This process is illustrated by the following image (note that we do not claim that nodes $a_{i,j}$ are in fact mutually distinct nodes from our model).



Note now that since square is also true at $a_{0,1}$, then $a_{0,1}$ must satisfy anticlockwise. Since going R and then U from $a_{0,1}$ takes us to $a_{1,2}$ and since $\texttt{when}(R \cdot U, \texttt{loop}(L \cdot R))$ is true at $a_{0,1}$, there is some node which we label $a_{0,2}$, that is reached by traversing an L-labelled edge from $a_{1,2}$. Note that this also implies that there is an R-labelled edge from $a_{0,2}$ to $a_{1,2}$. Again, since $\texttt{when}(R \cdot U \cdot L, \texttt{loop}(D \cdot U))$ is true at $a_{0,1}$ and $a_{0,2}$ can be reached by $R \cdot U \cdot L$ we have that there is a point $a'_{0,1}$ connected to $a_{0,2}$ by an D-labelled edge (and in the other direction by an U-labelled one). But now since $a_{0,1}$ also satisfies $\texttt{loop}(R \cdot U \cdot L \cdot D)$ and $a'_{0,1}$ is reached from $a_{0,1}$ by a path labelled $R \cdot U \cdot L \cdot D$,we have that $a'_{0,1} = a_{0,1}$. Thus we can draw a square starting in $a_{0,1}$, going in anticlockwise direction. This is illustrated in the following image.



We now note that with each edge there is a corresponding edge in the other direction with the appropriate label (e.g. L and R). To see this observe that in e.g. $a_{0,0}$ we have that $\texttt{loop}(U \cdot D)$ is true. This means that there is an U-edge from $a_{0,0}$ to $a_{0,1}$ and also an D-edge from $a_{0,1}$ to $a_{0,0}$ and analogously for all other edges.

In particular there is an R-edge from $a_{0,0}$ to $a_{1,0}$, so we can also complete the clockwise square started at $a_{1,0}$ and continuing through $a_{1,1}$ and $a_{2,1}$. This is done by the means of formula clockwise.

It is straightforward to see that this process can be continued for any number of steps, starting from the main diagonal and completing the squares above the diagonal in an anticlockwise direction, while completing the ones below the diagonal in a clockwise direction. Thus we showed that we can force a square grid by our formula.

Define now $t(i, j) = T_l$, where α_l is the unique formula of the form $\langle a^l \cap \varepsilon \rangle$ that is true at any point $a_{i,j}$ by means of γ_2. Note that γ_2 also forces the tiling t to be proper, since the formula tile assures that the tile $t(i+1, j)$ and $t(i, j+1)$ can only come from the set of tiles compatible with $t(i, j)$ in the appropriate direction.

Thus we have shown that if formula $\gamma_{\mathcal{T}}$ is satisfiable, then $\mathcal{T}$ can tile the positive plane $\mathbb{N}\times\mathbb{N}$. This implies that the set $\{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ is disjoint from S_{notiling}.

On the other hand, suppose that $\mathcal{T} = \{T_1,\ldots,T_k\}$ can tile the plane periodically, that is it can tile the torus $\mathbb{Z}_n \times \mathbb{Z}_m$ for some integers n and m. Let t be the tiling function $t : \mathbb{Z}_n \times \mathbb{Z}_m \to \mathcal{T}$ that witnesses this periodic tiling. We define the graph database G containing at most $(n+1)\cdot(m+1)+(k-2)$ nodes and satisfying $\gamma_{\mathcal{T}}$ as follows.

First, let

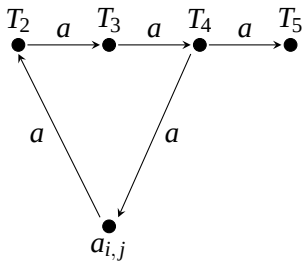
$$V = \{a_{i,j} : i = 1,\ldots,n+1 \text{ and } j = 1,\ldots,m+1\} \cup \{T_2,\ldots,T_k\}.$$

Next add the following edges to our graph.

1. For vertical edges:
   - for $i = 1\ldots n+1$ and $j = 1\ldots m$ put an U-edge between $a_{i,j}$ and $a_{i,j+1}$ and an D-labelled one in the other direction;
   - for $i = 1\ldots n+1$ put an U-labelled edge between $a_{i,m+1}$ and $a_{i,1}$ and an D-labelled one in the other direction.
2. Analogously for horizontal edges:
   - for $i = 1\ldots n$ and $j = 1\ldots m+1$ put an R-edge between $a_{i,j}$ and $a_{i+1,j}$ and an L-labelled one in the other direction;
   - for $j = 1\ldots m+1$ put an R-labelled edge between $a_{n+1,j}$ and $a_{1,j}$ and an L-labelled one in the other direction.

Also, define $T_2, T_3, \ldots, T_k$ to form an a-labelled chain. That is we add an a-edge between T_i and T_{i+1}, for $i = 2,\ldots k-1$.

Next, for each $a_{i,j}$,where $i \neq n+1$ and $j \neq m+1$ let T_l be the unique tile given by the tiling $t(i,j)$. If $l = 1$ we add an a-edge from $a_{i,j}$ to itself. If $l > 1$ we add an a-labelled edge from $a_{i,j}$ to T_2 and another a-labelled edge from T_l to $a_{i,j}$. This will allow us to satisfy the formula $\alpha_i = \langle a^l \cap \varepsilon\rangle$ as illustrated in the following image.



Finally, for $i = n+1$ and $j \neq m+1$ let $T_l = t(1, j)$ and define the outgoing a-edges from $a_{n+1,j}$ to T_2 and from T_l as above. Similarly, for $i \neq n+1$ and $j = m+1$ do the same for $T_l = t(i, 1)$. Lastly, repeat the procedure for $a_{n+1,m+1}$ and $T_l = t(1, 1)$.

Consider now formula γ_1. Note that we can reach any point by using U and R transitions, so we have to check that `square` is true at any point. But this is straightforward to check, since our graph G is a simple finite grid that folds onto itself (that is from each point on the edge we can continue in the appropriate direction). The fact that γ_2 is true follows from the fact that t is a periodic tiling. Namely, at any point in the graph G, precisely one α_i is true (note that we require the a-path to loop over the node, so only one such path exists by our construction). After that, any R or U step we take will take us to a node where the appropriate β_j or β^j is true since t is a tiling.

This shows that the set $S = \{\varphi \mid \exists G \text{ s.t. } [\![\varphi]\!]^G \neq \emptyset\}$ contains the set $\{\gamma_{\mathcal{T}} \mid \mathcal{T} \in \mathcal{S}_{\text{period}}\}$. As mentioned above, Fact 10.2.2 implies that the set of all satisfiable GXPath node formulas S, is not recursive.

In particular this implies that query containment for GXPath is not decidable, since the latter would entail recursivity of the set S by simply checking does the containment $[\varphi] \subseteq [\neg\top]$ hold.

Thus we proved that query containment for GXPath is undecidable, even with a fixed alphabet Σ of edge labels. □

Note that the previous theorem also implies undecidability of query containment for TriAL*, since the language was shown to contain GXPath in Section 9.3.

Corollary 10.2.3. *Query containment for* TriAL* *is undecidable.*

Due to the before mentioned connection of GXPath to PDL, we have a result on satisfiability of PDL with negation over finite models.

Corollary 10.2.4. *The satisfiability problem for PDL with negation on paths is undecidable over finite models, even in the absence of propositional variables.*

In fact, by carefully examining the proof, one can check that the use of negation is quite limited and that we only use intersection and the fact that GXPath$_{\text{reg}}$ can define the set of all pairs of mutually different nodes via the expression $\overline{\varepsilon}$. We are hoping that further adaptations of the proof could lead to solving the well know open problem of finite satisfiablity for PDL formulas with intersection [Göller et al., 2009].

As in the previous sections, we have the following question: what are the restrictions on GXPath$_{\text{reg}}$ that make containment decidable? The most natural candidates are of course the ones that forbid negation. Since we have two forms of negation, one on node formulas and another on path formulas, we consider both GXPath$_{\text{reg}}^{\text{pos}}$ and GXPath$_{\text{reg}}^{\text{path-pos}}$, the positive navigational fragments of GXPath.

Note that, as opposed to the classes from previous sections, the word "positive" refers here to restrictions of navigational properties, and not of data manipulation abilities.

Using the equivalence of $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$ and NREs (see Theorem 9.2.3) we can use the result on containment of NREs from [Reutter, 2013a] to obtain the following.

Proposition 10.2.5 ([Reutter, 2013a])**.** *The decision problem* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}$*) is* PSPACE*-complete.*

Exploiting connections with PDL, we obtain the following result for $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$.

Theorem 10.2.6. *The decision problem* CONTAINMENT *(*$\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$*) is* EXPTIME*-complete.*

Proof. To show the upper bound we first prove that the problem of query containment for $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ path formulas can be polynomially reduced to the problem of satisfiability of $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ node formulas. The idea is similar to the one used in [ten Cate and Lutz, 2009] to show that the two problems are inter-reducible for XPath queries on trees.

Let α and β be two $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ path formulas and let Γ denote the alphabet of all symbols occurring in α and β plus one additional symbol b. It is straightforward to see that if α is not contained in β, then there is a graph G witnessing this non-containment that uses labels from Γ only. (The idea here is that only labels appearing in α and β are relevant, and all the other labels can be replaced by the new label.)

Let now $\Gamma' := \Gamma \times \{0,1\}$. That is, Γ' contains copies of each label decorated with either 0 or 1. We define α' as a formula obtained from α by replacing each occurrence of a label a by $(a,0) \cup (a,1)$ and likewise for β'. Finally, let $\mathtt{out}$ be the formula $\bigcup_{a\in\Gamma}(a,1)$. We show that α is contained in β if and only if the formula

$$\varphi := \langle \alpha'[\mathtt{out}]\rangle \wedge \neg\langle \beta'[\mathtt{out}]\rangle$$

is not satisfiable.[1]

Assume first that α is not contained in β. Then there is a graph G and two nodes $v, v' \in G$ such that $(v,v') \in [\![\alpha]\!]^G$, but $(v,v') \notin [\![\beta]\!]^G$. As mentioned above, we can assume, without the loss of generality, that G uses only labels from Γ. Define now G' to be a Γ' labelled graph where each label a is replaced by $(a,0)$. In addition, we also add a loop from v' to v' labelled $(b,1)$. Since v' is the only node with an outgoing edge whose label has second component equal to 1 we get that $v \in [\![\varphi]\!]^{G'}$, as required.

On the other hand, assume that φ is satisfiable. Let G' be any graph such that there is $v \in G'$ with $v \in [\![\varphi]\!]^{G'}$. Let G be a graph obtained from G' be replacing every edge labelled $(a,0)$ or $(a,1)$ by a (note that the b-edges can be thrown away, since neither α, nor β can access them).

[1] Note that here we are writing e.g. $[\alpha]$ instead of $[\langle\alpha\rangle]$, when checking that a node has an outgoing α-path.

Since $v \in [\![\varphi]\!]^{G'}$, there is some $v' \in G'$ such that $(v,v') \in [\![\alpha'[\mathtt{out}]]\!]^{G'}$. It is then straightforward to see that $(v,v') \in [\![\alpha]\!]^{G}$. On the other hand, if we had that (v,v') is in $[\![\beta]\!]^{G}$, then we would also get that $(v,v') \in [\![\beta'[\mathtt{out}]]\!]^{G'}$, (since v' must have an outgoing edge with second component equal to 1 to satisfy $\alpha'[\mathtt{out}]$), which contradicts the fact that $v \in [\![\varphi]\!]^{G'}$. Thus α is not contained in β, as required. (Note that it could still be the case that $v \in [\![\langle\alpha\rangle]\!]^{G}$ and $v \in [\![\langle\beta\rangle]\!]^{G}$, but we are interested in binary containment.)

We have thus shown that query containment for $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ path formulas is polynomially reducible to (un)satisfiability of node formulas of the same language. Using this and the fact that $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ is contained in PDL (in fact $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ is the same as PDL without variables) we can use the decision procedure for PDL to solve $\mathsf{GXPath}_{\mathrm{reg}}^{\text{path-pos}}$ query containment. Since the former is in ExpTime (see [Harel et al., 2000], Theorem 8.4), we obtain the desired result.

The lower bound follows from adapting known ExpTime-complete results regarding the satisfiability of PDL versions close to XPath (see e.g. Section 4.4 of [Alechina et al., 2003]; or Theorem 8.4 in [Harel et al., 2000]). These results present reductions from the acceptance problem of a Turing machine that decides a language in ExpTime. The only difficulty in the adaptation of these proofs is dealing with a bounded alphabet, since the natural adaptation of these results would result in a reduction needing an unbounded alphabet. But this can be done by coding the symbols of the alphabet as binary strings—of unbounded length but now using a bounded alphabet—as it is repeatedly done in [Barceló et al., 2013b] (see the ExpSpace-hardness proof). For example, if Σ contains 4 characters, then we treat them as strings 00, 01, 10 and 11. □

10.2.2 Containment with data values

We will now consider how data value tests affect containment of GXPath queries. Recall from Chapter 7 that these are either of the form $\alpha_{=}, \alpha_{\neq}$, with α being a path expression, or $\langle \alpha = \beta \rangle, \langle \alpha \neq \beta \rangle$ (as mentioned previously here we will disregard constants). The first type of tests is denoted with $\sim$, while the second is denoted with eq. These can again be coupled with positive navigational features restricting negation in node or path formulas, giving rise to six different fragments, ranging from $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\mathsf{eq})$ to $\mathsf{GXPath}_{\mathrm{reg}}(\sim)$.

To examine their containment problem, notice first that it was shown in Chapter 9, that even $\mathsf{GXPath}_{\mathrm{reg}}^{\mathrm{pos}}(\sim)$ contains RQDs. Theorem 10.1.7 then implies that containment for all of the fragments with $\sim$ tests is undecidable. From Theorem 10.2.1 we also get undecidability of $\mathsf{GXPath}_{\mathrm{reg}}(\mathsf{eq})$. We summarise these results in the following corollary.

Corollary 10.2.7. *The problems*

- Containment *($\mathsf{GXPath}_{reg}^{pos}(\sim)$)*,

- CONTAINMENT *($\mathsf{GXPath}_{reg}^{path\text{-}pos}(\sim)$) and*
- CONTAINMENT *($\mathsf{GXPath}_{reg}(\sim)$)*
- CONTAINMENT *($\mathsf{GXPath}_{reg}(eq)$)*

are undecidable.

The next step in the search for decidable fragments of GXPath would be to restrict data tests to equality only (i.e. forbid subexpressions of the form $\alpha_{\neq}$ and similarly for eq tests). Note that these were already introduced in Section 7.4. Here we use $\sim_=$ to denote fragments using only $\alpha_=$ tests and $\mathsf{eq}_=$ for fragments using only $\langle \alpha = \beta \rangle$. From Theorem 10.2.1 we already know that containment for $\mathsf{GXPath}_{reg}(\sim)$ with such restriction is undecidable. However, results for similar fragments of RQDs give some hope that containment for e.g. $\mathsf{GXPath}_{reg}^{path\text{-}pos}(\sim_=)$ and $\mathsf{GXPath}_{reg}^{pos}(\sim_=)$ with such restrictions might be decidable. We summarise known results in the following image. Note that the fragments are positioned in a way that reflects their relative expressive power (see Section 7.4).

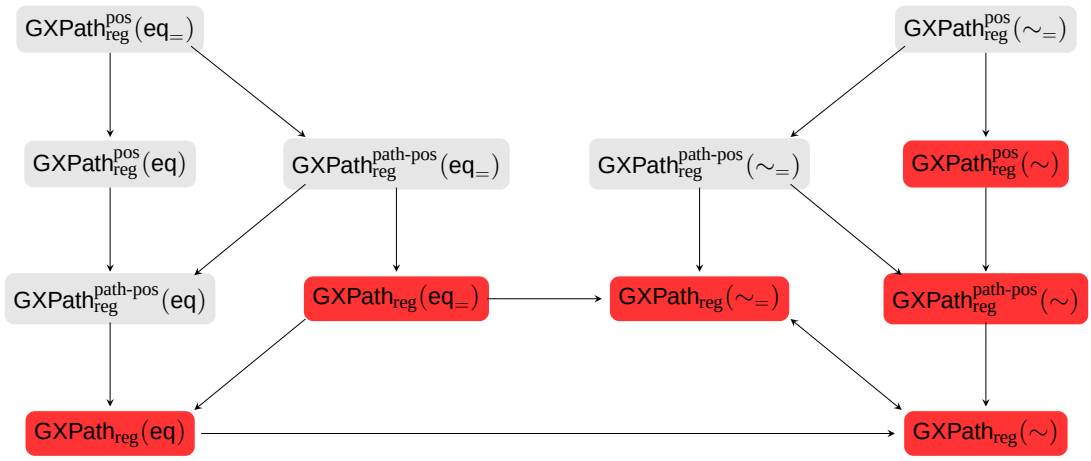


Figure 10.2: Containment problem for GXPath_{reg} fragments with data value tests. Red colour indicates undecidability. Grey colour indicates that the status of containment problem is still unknown.

10.2.3 Coming back to the core

When traditional XPath over trees is considered, negative results about query containment, can often be surpassed [Schwentick, 2004, Figueira, 2010b] by restricting attention to the core fragment allowing Kleene star to range only over basic navigational axes. It therefore makes sense to see how this restriction is reflected over graphs where a hierarchy of GXPath_{core} fragments, analogous to the one from Figure 10.2 exists.

By carefully examining the proof of Theorem 10.2.1 we can see that all of the expressions used there in fact belong to $\mathsf{GXPath}_{\mathrm{core}}$, therefore implying undecidability of all fragments using negation both on node and path formulas. In the following figure we summarise the known results about containment of core fragments of GXPath with various data tests.

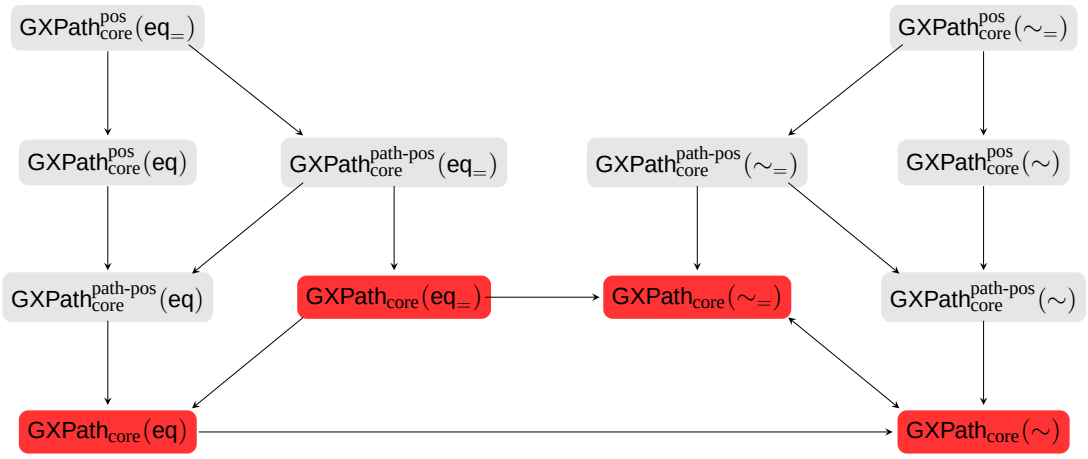


Figure 10.3: Containment problem for $\mathsf{GXPath}_{\mathrm{core}}$ fragments with data value tests. Red colour indicates undecidability. Grey colour indicates that the status of containment problem is still unknown.

Here we see that, similarly as with $\mathsf{GXPath}_{\mathrm{reg}}$ there are still many unresolved questions and a further study into the problem is warranted. Note that with core fragments, even when navigation alone is considered, we can no longer rely on standard tools from automata theory or formal languages, since the expressive power is severely restricted. This makes the fragments more likely to have decidable containment problem, but the search for correct bounds seems to be a challenging task in the same manner as it was for XPath over trees [Figueira, 2010b].

10.3 Summary of containment results

After conducting an initial study of query containment for main classes of queries for graphs with data, we conclude that the picture here is quite different from the one for traditional navigational languages. In particular, there is a sharp contrast between RPQs or CRPQs, where containment is decidable, and any of the known extension of RPQs that handle data values. Undecidability for the class of RQMs comes as not a surprise, due to high complexity of query evaluation and powerful data manipulation mechanism, but we have seen that even classes with good query evaluation properties can have undecidable containment.

The presence of inequality tests seems to be one of the major detractors here, although the ability to define complex navigational patterns can lead to undecidability as well. Thus, it

Data comparisions	RQD	RQM	2RQD	2RQM	$\mathsf{GXPath}_{reg}^{pos}(\sim)$	$\mathsf{GXPath}_{reg}^{path\text{-}pos}(\sim)$	$\mathsf{GXPath}_{reg}(\sim)$
none	PSPACE-c*		PSPACE-c*		PSPACE-c*	EXPTIME-c	und.
full	und.	und.	und.	und.	und.	und.	und.
positive	PSPACE-c	EXPSPACE-c	?	und.	?	?	und.

Table 10.1: Complexity of containment of data graph queries. Some classes have synonyms, not given for clarity: i.e. RQDs and RQMs with no data comparisons are RPQs. Results, known before, are marked with '*', "-c" stands for "complete".

seems that to obtain decidable fragments one has to limit attention to purely positive subclasses. The situation further complicates in the presence of inverse operator. We summarise results for main classes of queries in Table 10.1.

All of this shows that, although most of graph query languages are already well established, there is still some fine tuning needed to define languages with desirable static analysis properties. While results on query containment are well understood for path queries introduced in Chapter 4, there are still some gaps when it comes to graph languages. In particular, we would like to fully understand the containment problem for all fragments of GXPath. Some results in previous sections give us hope that decidability could be obtained for positive fragments using only equality tests and for core fragments.

In particular, the decidability of containment for equalities-only versions of $\mathsf{GXPath}_{reg}^{pos}$ and $\mathsf{GXPath}_{reg}^{path\text{-}pos}$ is still open. Furthermore, the picture for classes that use eq data tests is also not well understood (Figure 10.2), and for core fragments we have only started to scratch the surface (Figure 10.3). Another valid line of research is also to purse decidable fragments of TriAL, where some initial work was done, albeit for much more restricted languages [Rudolph and Krötzsch, 2013].

All of this shows that query containment for graph languages promises to be a fruitful direction for future research, hopefully leading to development of many new techniques as was the case with XML [Figueira, 2010b].

Part IV

Wrapping up

Chapter 11

Conclusions and future work

Historically querying graph data was done in two completely separate ways: either one would query the raw data residing in the graph while completely disregarding how the data is connected, or one would query only the topology of the model, determining intricate patterns connecting the data, but not doing any reasoning on the data itself. The main objective of this dissertation was to explore principles of good query language design that combines these two modes of querying. Namely, we propose languages that, in addition to being able to ask questions about the underlying topology of the model, also allow to determine how the actual data changes while navigating the graph.

In order to do so we study how adding various data manipulation features and mixing them with navigational capabilities of the language at hand affects the complexity of main reasoning tasks and how it relates to the expressive power of the language. In this thesis we proposed two classes of languages: path languages and graph languages, based on the set of basic navigational features they allow. Path languages extend the basic RPQs with different data manipulation capabilities and here we see that efficiency of each one of them, as well as their expressive power, is closely related to the nature of data tests we allow. Although navigationally quite simple (namely they can describe only paths), when extended with the ability to store and compare data values, they become a powerful language for reasoning about graphs. This power comes with a price though, as the complexity of query evaluation is relatively high (although no worse than for traditional relational languages) and basic static aspects of the language, such as containment or satisfiability, quickly become undecidable. Restrictions are, of course, possible, but quite often the natural restrictions do not amount to any gain in efficiency, and cutting out the ability to store data in variables, while leading to highly efficient languages, results in somewhat limited expressive power. This is, of course, a fact one has to deal with, as even the basic matching of equal data values, such as the one used in well known grep expressions from Unix operating systems, results in intractable complexity of query evaluation.

Graph languages on the other hand try to avoid this difficulty by allowing only simple

data value tests that were proven to be relevant in the context of XML (recall that our main graph language, GXPath, is based on the XML query language XPath), while at the same time allowing more intricate navigational patterns lying outside of scope of path languages. Since the language is highly efficient (namely query evaluation is always tractable), and since both the navigational and data manipulation abilities it allows were shown to be of interest to many practitioners, we believe that certain features of this language should be considered as a basic building block of any practical graph language. Some users, however, simply need the ability to store the data and check how it changes along the path, so to them path languages will have a greater appeal, despite the higher complexity. Another language, TriAL, that we introduced to query RDF documents, could be used to overcome this issue, but only to a certain extent, since it offers a bit more memory storage than GXPath and comes with only a slightly higher complexity of the query evaluation problem. However, as we discuss in the next section, it seems that users have to pick from one brand of languages, either path or graph, based on the type of queries they intend to ask and the availability of computational resources.

As one of the main goals of this study is to be able to pinpoint a specific set of primitives that a query language should posses in order to meet user requirements, in Section 11.1 we discuss how to chose the appropriate language and how such a choice can be balanced in terms of expressivity and efficiency. We conclude with some directions for future work in Section 11.2.

11.1 Choosing the right language

Having studied how various data and navigational features affect the ability of the language to express relevant queries, as well as how they influence efficiency, we come to a conclusion that there are no clear winners when it comes to choosing a particular language, if the context is not known. Indeed, as some groups of practitioners will value a certain set of functionalities above others, they will consider a language allowing these functionalities better suited for their purposes, thus making it a worthy candidate for their particular goal, while others might dismiss it on grounds of high complexity, or the inability to express the type of queries they find relevant. Because of this we can not bring one of the proposed languages forward as *the* language for graphs with data, however, we can point to good candidates when a specific capability is required. Below we provide some recommendations of a suitable language if the user has a specific goal in mind.

Navigational queries In the past the main focus of graph languages has been on retrieving information about how the data is connected and not about the actual data. And while most modern systems now also include some sort of data handling capability, navigational query-

ing still forms the core of many languages, and they are often used to ask strictly navigational queries. If the users main concern are such queries then the answer to the question of which language to use is quite clear – it is GXPath or some of its many fragments or variants. Indeed, considering all of the languages proposed both here and in the research literature, it is difficult to find one that is both as expressive and as efficient in terms of query answering. On top of that, the language is closely connected to logic, both FO and PDL, and is capable of expressing queries outside of the scope of most previous recommendations (with the sole exception of extended RPQs [Barceló et al., 2012b], which are incomparable to GXPath, but also much less efficient). Therefore, as far as navigational queries are considered it seems that GXPath provides good balance between expressive power and efficiency and should be strongly considered as a core of any purely navigational language. Some of the issues come with respect to query evaluation, as it is not currently known if evaluation algorithms for GXPath can be parallelized. We leave this interesting question as one of the directions for future research. We would also like to note that from the point of view of static analysis the language fares somewhat worse than its competitors, but this is to be expected with such high expressivity. Note that even then the most natural restrictions, still more powerful than the previously proposed languages, again regain good algorithmic properties of query containment and satisfiability. Overall we believe that, despite these minor difficulties which one still has to overcome even with much simpler languages, GXPath can be recommended as the navigational standard for graphs.

Hybrid languages Although navigational queries are important in and of themselves, the true power of the graph data model lies in its ability to mix navigation and the data. However, since this dissertation is a first detailed study of languages that allow such mixing, it is still not clear as to which language should be chosen above all others. Indeed, it seems that different requirements call for different design principles to be applied to the language, all of them with their strengths and weaknesses, but that no entirely uniform approach can be taken. This is, of course, not so unusual for an area in its infancy and hopefully with the maturation of the field it will become apparent how particular data manipulation tasks can be pruned to establish a good querying basis that can be added to the navigational part of the language. In the mean time, we provided several good options, that can, as we discuss next, be used to meet specific requirements that a group of users might impose.

Languages with memory When memory usage is required, for example to ask queries that propagate data (in)equality along the path connecting two data points, it seems that *RQM*s are the way to go. Not only do these queries have high expressive power matching that of register automata, but their syntax is also clear and easily understandable. Furthermore, they can easily be extended to allow backward navigation and conjunction, making them a desirable candidate for the user to chose. Of course, if strict scoping rules that mimic the use of variables

in usual programming languages are required, then we turn to *RQB*s, where, with a small hit to expressive power, we still retain all of the desirable properties of *RQM*s. On the other hand, if we only need to use memory to match same data items in multiple locations we can use variable automata, or some of their restrictions. In all of these cases the proposed language has a great deal of expressive power when it comes to data manipulation, allowing us to store and compare data values as one would in any common programming language, although their navigational base does not extend beyond that of ordinary RPQs. The price we have to pay for this expressive power comes in terms of high complexity of basic algorithmic tasks such as query evaluation and query containment. The evaluation problem is PSPACE-complete for *RQM*s and *RQB*s and it is well known that the best we can do with such approach, even if we remove several capabilities from models such as variable automata, is *NP* (see [Aho, 1990]).

Highly efficient languages To overcome the issue of high complexity we first introduced the class of *RQD*s. These queries, although still being able to express many interesting properties of graphs, are somewhat limited, and as we showed, one obtains the same bounds for query evaluation even for the navigationally much richer language of GXPath. Furthermore, although the data tests used in GXPath are based on the same idea as the ones in *RQD*s, combining them with the ability to define patterns and not only paths (as in the case of *RQD*s), allows us to subsume XPath-style data tests that have been tried and tested by XML practitioners. Finally, the language has a very clean logical core – namely it is equivalent to $(\mathrm{FO}^*)^3(\sim)$, the three variable fragment of first-order logic with binary transitive closure and data value comparisons. All of this leads to the conclusion that when high efficiency is sought GXPath with data value comparisons seems to be the most likely candidate to pick. It is also worth emphasizing again that, in addition to being able to define data tests that were shown to be useful in practice, we also get the best possible language in terms of navigational features, and all of that basically for free – the complexity does not change much even when compared to RPQs that do not deal with data values. Of course, if the users require memory, they might find the language somewhat lacking, but as theoretical results tell us, to use memory freely (Theorem 4.2.7), or even in a severely restricted way (Theorem 4.5.6), we have to pay the price in terms of efficiency (after all, if expressing a certain property is *NP*-hard it is *NP*-hard and there is no way around this fact). Overall, GXPath seems to be a strong candidate when high efficiency is required and in the future research we would like to address the question of parrallelizability of the evaluation algorithms for the language, or in the case the problem is PTIME-hard (here we use the usual complexity assumption that $\mathrm{NC} \neq \mathrm{PTIME}$), to find fragments that make evaluation easier.

What to do when graph languages fail Finally, when the users require a language for querying a slightly more general model of RDF, we argued that the language of choice should be TriAL. Here one can, of course, use various graph languages, as was successfully done in the

past (for example NREs form the navigational basis of n-SPARQL [Pérez et al., 2010], while the latest standard of the SPARQL query language for RDF uses a variation CRPQs [Harris and Seaborne, 2013]). Another graph language that can be seen as useful for this is GXPath and particularly its conjunctive version, as it allows slightly more varied queries than NREs. However, as we showed in Chapter 8, applying graph languages to RDF will have some inherent limitations linked to it, as it disregards the fact that edge labels in RDF are nodes themselves. To overcome this issue we introduced the language TriAL, geared exclusively towards the RDF data model and allowing users to express many properties that lie outside of the scope of graph languages. The language also retains good evaluation bounds and its datalog counterpart provides us with an intuitive declarative syntax for the language, thus making it a good choice of a theoretical basis for querying RDF documents.

11.2 Where to go from here

This thesis initiated the study of query languages for graph with data, and while many questions are already resolved, there are still several questions remaining opened and, as with any area that is just beginning, many possible directions for future research. We would like to finish by briefly naming a few of them:

Practical issues The theoretical study that we undertook here enabled us to determine the practical potential of a query language. The next logical step is to efficiently implement these languages using the algorithms and reasoning procedures we developed and test how they behave in practice where the optimal theoretical solution might not always be what the users need. While doing these practical experiments we hope to interact with graph database vendors and suggest which features of a graph language are best suited for their specific goals and how to implement such features. The problem with the existing systems, such as [Dex, 2013, Neo4j, 2013, InfiniteGraph, 2013], is that the syntax and semantics that they use is not precisely defined, which makes it difficult to understand where the main issues that practitioners face when using such products originate from. On top of that, most systems fail to express many important graph queries that mix topological properties and data. What we hope to produce is a good library of procedures that vendors could use to efficiently implement various features needed in practice.

Note that this is a difficult and challenging task which promises to lead to many new interesting research topics, such as the issue of storing and indexing graph data, and particularly its performance on massively parallelizable systems.

Additional features We have already explored how some basic add-ons, such as inverses and conjunctive queries, affect the language. There are, of course, many other features that

come into play when languages are applied, such as aggregation or allowing more freedom in manipulating the attribute data. For example we could compare string values for substrings, or do arithmetical operations over integers. It would therefore be interesting to look how adding such features can be accommodated into the languages we proposed in this thesis. What we also hope to achieve is a syntax that would be more attractive to users who require multiple attributes per node (or edge). There are various options that present themselves here, as our languages are readily extendible to support this functionality, but some careful examination of actual requirements by various groups of users is needed to determine which syntax is better suited for such a language.

Using languages in different scenarios Connected to the practical considerations above we would also like to explore how our languages can be used in new application domains that require navigational and data patterns to be detected in the underlying model.

The first area we would like to tackle is querying of the Semantic Web, where SPARQL seems to be the current language of choice. What we propose is testing if a more "lightweight" language, namely the conjunctive version of GXPath would do the trick. We already know that from a theoretical point of view evaluation is more efficient in this language and there are several important queries that SPARQL can not express that our language can. Of course, our language also does not capture all of SPARQL, and it would therefore be interesting to see if conjunctive GXPath is sufficient to express most queries that are of interest to practitioners.

The second area we had in mind is querying data and workflow provenance. Here one typically stores information about how data is created and modified and sometimes it is useful to have the ability to track the origins of such data. For example if a bug is found in a large software project it is important to locate the library, or the modification of code, that led to this bug. One language that naturally lends itself to such queries is TriAL and we are hoping to see how its implementations fare in practice, especially considering the fact that the queries such as the one above are often outside the reach of languages that are currently used to extract information about such data.

Static analysis When considering static properties of our languages we mainly focused on containment, but there are several other important questions to consider here. For example to optimize queries one often uses equivalence and satisfiability is often crucial for checking consistency of documents. It would therefore be interesting to explore these properties for the languages we proposed in previous chapters. On top of that, there are also many open questions relating to containment, particularly when various fragments of GXPath are considered, all of these promising to form a fruitful direction for future research.

Incomplete information Finally, it would be interesting to see how missing and incomplete data impacts graph languages. To an extent this problem has been previously addressed in [Reutter, 2013b, Barceló et al., 2014], however, there only navigational aspects of graph languages were taken into account, and data values were not considered. The situation when data values are present (or, as we are dealing with incomplete information, missing) seems to complicate the issue quite considerably and promises to hold many intricate problems that need to be tackled.

Bibliography

[Abiteboul et al., 1999] Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kauffman.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Abiteboul et al., 1997] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88.

[Abiteboul and Vianu, 1999] Abiteboul, S. and Vianu, V. (1999). Regular path queries with constraints. *J. Comput. Syst. Sci.*, 3(58):428–452.

[Aho, 1990] Aho, A. V. (1990). *Handbook of Theoretical Computer Science*, chapter Algorithms for finding patterns in strings. MIT Press.

[Alechina et al., 2003] Alechina, N., Demri, S., and de Rijke, M. (2003). A modal perspective on path constraints. *J. Log. Comput.*, 13(6):939–956.

[Alechina and Immerman, 2000] Alechina, N. and Immerman, N. (2000). Reachability logic: An efficient fragment of transitive closure logic. *Logic Journal of the IGPL*, 8(3):325–337.

[Amer-Yahia et al., 2009] Amer-Yahia, S., Lakshmanan, L. V. S., and Yu, C. (2009). SocialScope: Enabling Information Discovery on Social Content Sites. In *CIDR*.

[Anand et al., 2010] Anand, M. K., Bowers, S., and Ludäscher, B. (2010). Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, pages 287–298.

[Andréka et al., 2001] Andréka, H., Németi, I., and Sain, I. (2001). *Handbook of Philosophical Logic*, volume 2, chapter Algebraic logic. Springer, 2 edition.

[Angles, 2012] Angles, R. (2012). A comparison of current graph database models. In *ICDE Workshops*, pages 171–177.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1).

[Anyanwu and Sheth, 2003] Anyanwu, K. and Sheth, A. (2003). ρ-queries: enabling querying for semantic associations on the semantic web. In *12th International World Wide Web Conference (WWW)*, pages 690–699.

[Arenas and Pérez, 2011] Arenas, M. and Pérez, J. (2011). Querying semantic web data with SPARQL. In *PODS*, pages 305–316.

[Bachman, 1973] Bachman, C. W. (1973). The Programmer as Navigator. ACM Turing Award lecture. *Communications of the ACM*, 16(11):653–658.

[Barceló, 2013] Barceló, P. (2013). Querying graph databases. In *32th ACM Symposium on Principles of Database Systems (PODS)*.

[Barceló et al., 2012a] Barceló, P., Figueira, D., and Libkin, L. (2012a). Graph logics with rational relations and the generalized intersection problem. In *27th Annual IEEE Symposium on Logic in Computer Science (LICS)*.

[Barceló et al., 2012b] Barceló, P., Libkin, L., Lin, A. W., and Wood, P. T. (2012b). Expressive languages for path queries over graph-structured data. *ACM TODS*, 38(4).

[Barceló et al., 2011] Barceló, P., Libkin, L., and Reutter, J. (2011). Querying graph patterns. In *30th ACM Symposium on Principles of Database Systems (PODS)*, pages 199–210.

[Barceló et al., 2014] Barceló, P., Libkin, L., and Reutter, J. (2014). Querying regular graph patterns. *Journal of the ACM*, 61(1).

[Barceló et al., 2012c] Barceló, P., Pérez, J., and Reutter, J. (2012c). Relative expressiveness of nested regular expressions. In *AMW*, pages 180–195.

[Barceló et al., 2013a] Barceló, P., Pérez, J., and Reutter, J. L. (2013a). Schema mappings and data exchange for graph databases. In *ICDT*.

[Barceló et al., 2013b] Barceló, P., Reutter, J. L., and Libkin, L. (2013b). Parameterized regular expressions and their languages. *Theor. Comput. Sci.*, 474:21–45.

[Benedikt et al., 2008] Benedikt, M., Fan, W., and Geerts, F. (2008). Xpath satisfiability in the presence of dtds. *Journal of the ACM*, 55(2).

[Benedikt and Koch, 2008] Benedikt, M. and Koch, C. (2008). Xpath leashed. *ACM Computing Surveys (CSUR)*, 41(1).

[Bienvenu et al., 2013] Bienvenu, M., Ortiz, M., and Šimkus, M. (2013). Conjunctive regular path queries in lightweight description logics. In *IJCAI*.

[Bojanczyk, 2010] Bojanczyk, M. (2010). Automata for data words and data trees. In *RTA*.

[Bojanczyk et al., 2011] Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., and Segoufin, L. (2011). Two-variable logic on words with data. *ACM TOCL*, 12(4).

[Bojanczyk and Lasota, 2010] Bojanczyk, M. and Lasota, S. (2010). An extension of data automata that captures XPath. In *25th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 243–252.

[Bojanczyk et al., 2009] Bojanczyk, M., Muscholl, A., Schwentick, T., and Segoufin, L. (2009). Two-variable logic on data trees and XML reasoning. *Journal of the ACM*, 56(3).

[Bojanczyk and Parys, 2011] Bojanczyk, M. and Parys, P. (2011). Xpath evaluation in linear time. *J. ACM*, 58(4).

[Börger et al., 1997] Börger, E., Gräedel, E., and Gurevich, Y. (1997). *The Classical Decision Problem*. Perspectives in Mathematical Logics. Springer-verlag.

[Bouajjani et al., 2003] Bouajjani, A., Habermehl, P., and Mayr, R. (2003). Automatic verification of recursive procedures with one integer parameter. *Theoretical Computer Science*, 295.

[Bouyer et al., 2001] Bouyer, P., Petit, A., and Thérien, D. (2001). An algebraic characterization of data and timed languages. In *CONCUR*, pages 248–261.

[Calvanese et al., 2000] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. (2000). Containment of conjunctive regular path queries with inverse. In *7th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185.

[Calvanese et al., 2003] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. (2003). Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92.

[Calvanese et al., 2009] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. (2009). An automata-theoretic approach to regular XPath. In *DBPL*, pages 18–35.

[Calvanese et al., 2001] Calvanese, D., De Giacomo, G., Lenzerini, M., and Vardi, M. Y. (2001). View-based query answering and query containment over semistructured data. In *DBPL*, pages 40–61.

[Cassidy, 2003] Cassidy, S. (2003). Generalizing XPath for directed graphs. In *Extreme Markup Languages*.

[Chandra and Merlin, 1977] Chandra, A. and Merlin, P. (1977). Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90.

[Cleaveland and Steffen, 1993] Cleaveland, R. and Steffen, B. (1993). A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147.

[Consens and Mendelzon, 1990] Consens, M. and Mendelzon, A. (1990). Graphlog: A visual formalism for real life recursion. In *9th ACM Symposium on Principles of Database Systems (PODS)*, pages 404–416.

[Consens and Mendelzon, 1989] Consens, M. P. and Mendelzon, A. O. (1989). Expressing structural hypertext queries in graphlog. In *Hypertext*, pages 269–292.

[Cruz et al., 1987] Cruz, I., Mendelzon, A., and Wood, P. (1987). A graphical query language supporting recursion. In *ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD)*, pages 323–330.

[Cudré-Mauroux and Elnikety, 2011] Cudré-Mauroux, P. and Elnikety, S. (2011). Graph data management systems for new application domains. *PVLDB*, 4(12):1510–1511.

[David et al., 2013] David, C., Gheerbrant, A., Libkin, L., and Martens, W. (2013). Containment of pattern-based queries over data trees. In *ICDT*, pages 201–212.

[Demri and Lazić, 2009] Demri, S. and Lazić, R. (2009). Ltl with the freeze quantifier and register automata. *ACM TOCL*, 10(3).

[Demri et al., 2007] Demri, S., Lazić, R., and Nowak, D. (2007). On the freeze quantifier in constraint ltl: Decidability and complexity. *Information and Computation*, 205(1):2–24.

[Deutsch and Tannen, 2001] Deutsch, A. and Tannen, V. (2001). Optimization properties for classes of conjunctive regular path queries. In *8th International Workshop on Database Programming Languages (DBPL)*, pages 21–39.

[Dex, 2013] Dex (2013). DEX query language, Sparsity Technologies. http://www.sparsity-technologies.com/dex.php.

[Dey et al., 2013] Dey, S. C., Cuevas-Vicenttín, V., Köhler, S., Gribkoff, E., Wang, M., and Ludäscher, B. (2013). On implementing provenance-aware regular path queries with relational query engines. In *EDBT/ICDT Workshops*, pages 214–223.

[Fan, 2012] Fan, W. (2012). Graph pattern matching revised for social network analysis. In *ICDT*, pages 8–21.

[Fan et al., 2010a] Fan, W., Li, J., Ma, S., Tang, N., and Wu, Y. (2010a). Graph pattern matching: from intractable to polynomial time. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):264–275.

[Fan et al., 2011] Fan, W., Li, J., Ma, S., Tang, N., and Wu, Y. (2011). Adding regular expressions to graph reachability and pattern queries. In *27th International Conference on Data Engineering (ICDE)*, pages 39–50.

[Fan et al., 2010b] Fan, W., Li, J., Ma, S., Wang, H., and Wu, Y. (2010b). Homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):1161–1172.

[Fernández et al., 2000] Fernández, M. F., Florescu, D., Levy, A. Y., and Suciu, D. (2000). Declarative specification of web sites with strudel. *VLDB J.*, 9(1):38–55.

[Figueira, 2009] Figueira, D. (2009). Satisfiability of downward XPath with data equality tests. In *28th ACM Symposium on Principles of Database Systems (PODS)*, pages 197–206.

[Figueira, 2010a] Figueira, D. (2010a). Forward-XPath and extended register automata on data-trees. In *ICDT*, pages 231–241.

[Figueira, 2010b] Figueira, D. (2010b). *Reasoning on words and trees with data*. PhD thesis, ÉNS de Cachan.

[Figueira and Segoufin, 2009] Figueira, D. and Segoufin, L. (2009). Future-looking logics on data words and trees. In *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09)*, volume 5734 of *Lecture Notes in Computer Science*, pages 331–343. Springer.

[Figueira and Segoufin, 2011] Figueira, D. and Segoufin, L. (2011). Bottom-up automata on data trees and vertical XPath. In *28th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 93–104.

[Fletcher et al., 2011] Fletcher, G. H. L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S., and Wu, Y. (2011). Relative expressive power of navigational querying on graphs. In *ICDT*, pages 197–207.

[Fletcher et al., 2012] Fletcher, G. H. L., Gyssens, M., Leinders, D., Van den Bussche, J., Van Gucht, D., Vansummeren, S., and Wu, Y. (2012). The impact of transitive closure on the boolean expressiveness of navigational query languages on graphs. In *FoIKS*, pages 124–143.

[Florescu et al., 1998] Florescu, D., Levy, A. Y., and Suciu, D. (1998). Query containment for conjunctive queries with regular expressions. In *PODS*, pages 139–148.

[Fortune et al., 1980] Fortune, S., Hopcroft, J., and Wyllie, J. (1980). The directed homeomorphism problem. *Theoretical Computer Science*, (10):111–121.

[Freydenberg and Schweikardt, 2011] Freydenberg, D. and Schweikardt, N. (2011). Expressiveness and static analysis of extended conjunctive regular path queries. In *5th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*.

[Glaister and Shallit, 1996] Glaister, I. and Shallit, J. (1996). A lower bound technique for the size of nondeterministic finite automata. *Information Processing Letters*, 59(2):75–77.

[Goldblatt and Jackson, 2012] Goldblatt, R. and Jackson, M. (2012). Well structured program equivalence is highly undecidable. *ACM Trans. Comput. Log.*, 13(3).

[Göller et al., 2009] Göller, S., Lohrey, M., and Lutz, C. (2009). Pdl with intersection and converse: satisfiability and infinite-state model checking. *J. Symb. Log.*, 74(1):279–314.

[Gottlob et al., 2002] Gottlob, G., Grädel, E., and Veith, H. (2002). Datalog lite: a deductive query language with linear time model checking. *ACM TOCL*, 3(1):42–79.

[Gottlob and Koch, 2004] Gottlob, G. and Koch, C. (2004). Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113.

[Gottlob et al., 2005] Gottlob, G., Koch, C., and Pichler, R. (2005). Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491.

[Grädel, 1991] Grädel, E. (1991). On transitive closure logic. In *CSL*, pages 149–163.

[Gremlin, 2013] Gremlin (2013). Gremlin Language. https://github.com/tinkerpop/gremlin/wiki.

[Grumberg et al., 2010a] Grumberg, O., Kupferman, O., and Sheinvald, S. (2010a). Variable automata over infinite alphabets. In *Proceedings of the 4th International Conference on Language and Automata Theory and Applications (LATA)*, pages 561–572.

[Grumberg et al., 2010b] Grumberg, O., Kupferman, O., and Sheinvald, S. (2010b). Variable automata over infinite alphabets. Manuscript.

[Gupta and Mumick, 1995] Gupta, A. and Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18.

[Gurevich and Koryakov, 1972] Gurevich, Y. and Koryakov, I. (1972). Remarks on berger's paper on the domino problem. *Siberian Math. Journal*.

[Gutierrez et al., 2011] Gutierrez, C., Hurtado, C., Mendelzon, A. O., , and Pérez, J. (2011). Foundations of semantic web databases. *Journal of Computer and System Sciences*, 77(3):520–541.

[Gyssens et al., 1994] Gyssens, M., Paredaens, J., Van den Bussche, J., and Van Gucht, D. (1994). A graph-oriented object database model. *IEEE Trans. Knowl. Data Eng.*, 6(4):572–586.

[Harel et al., 2000] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. MIT Press.

[Harris and Seaborne, 2013] Harris, S. and Seaborne, A. (2013). SPARQL 1.1 query language. W3C recommendation. http://www.w3.org/TR/sparql11-query/.

[Hopcroft and Ullman, 1979] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company.

[Immerman and Kozen, 1989] Immerman, N. and Kozen, D. (1989). Definability with bounded number of bound variables. *IANDC*, 83(2):121–139.

[InfiniteGraph, 2013] InfiniteGraph (2013). Infinitegraph release 3.1 by objectivity inc. http://www.objectivity.com/infinitegraph.

[Ioannidis et al., 2011] Ioannidis, Y. E., Vayanou, M., Georgiou, T., Iatropoulou, K., Karvounis, M., Katifori, V., Kyriakidi, M., Manola, N., Mouzakidis, A., Stamatogiannakis, L., and Triantafyllidi, M. L. (2011). Profiling attitudes for personalized information provision. *IEEE Data Eng. Bull.*, 34(2):35–40.

[Jena, 2012] Jena (2012). The Apache Jena Manual. http://jena.apache.org.

[Jones, 1975] Jones, N. (1975). Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 1:68–75.

[Kaminski and Francez, 1994] Kaminski, M. and Francez, N. (1994). Finite memory automata. *Theoretical Computer Science*, 134(2):329–363.

[Kaminski and Tan, 2006] Kaminski, M. and Tan, T. (2006). Regular expressions for languages over infinite alphabets. *Fundamenta Informaticae*, 69(3):301–318.

[Kaminski and Tan, 2008] Kaminski, M. and Tan, T. (2008). Tree automata over infinite alphabets. In *Pillars of Computer Science*, pages 386–423.

[Kay, 2004] Kay, M. (2004). *XPath 2.0 Programmer's Reference*. Wrox.

[Klyne and Carroll, 2004] Klyne, G. and Carroll, J. J. (2004). RDF concepts and abstract syntax, W3C recommendation.

[Kostylev et al., 2014] Kostylev, E. V., Reutter, J. L., and Vrgoč, D. (2014). Containment of data graph queries. In *To appear in ICDT*.

[Lange, 2006] Lange, M. (2006). Model checking propositional dynamic logic with all extras. *J. Applied Logic*, 4(1):39–49.

[Lenzerini, 2002] Lenzerini, M. (2002). Data integration: a theoretical perspective. In *PODS*, pages 233–246.

[Leser, 2005] Leser, U. (2005). A query language for biological networks. *Bioinformatics*, 21(2):ii33–ii39.

[Libkin, 2004] Libkin, L. (2004). *Elements of Finite Model Theory*. Springer.

[Libkin et al., 2013a] Libkin, L., Martens, W., and Vrgoč, D. (2013a). Querying Graph Databases with XPath. In *ICDT*.

[Libkin et al., 2013b] Libkin, L., Reutter, J. L., and Vrgoč, D. (2013b). TriAL for rdf: Adapting graph query languages for rdf data. In *PODS*.

[Libkin et al., 2013c] Libkin, L., Tan, T., and Vrgoč, D. (2013c). Regular expressions with binding over data words for querying graph databases. In *DLT*.

[Libkin and Vrgoč, 2012a] Libkin, L. and Vrgoč, D. (2012a). Regular expressions for data words. In *LPAR*, pages 274–288.

[Libkin and Vrgoč, 2012b] Libkin, L. and Vrgoč, D. (2012b). Regular Path Queries on Graphs with Data. In *ICDT*, pages 74–85.

[Losemann and Martens, 2012] Losemann, K. and Martens, W. (2012). The complexity of evaluating path expressions in SPARQL. In *PODS*, pages 101–112.

[Martens, 2006] Martens, W. (2006). *Static Analysis of XML Transformation and Schema Languages*. PhD thesis, Universiteit Hasselt.

[Marx, 2003] Marx, M. (2003). Xpath and modal logics of finite dag's. In *TABLEAUX*, pages 150–164.

[Marx, 2005] Marx, M. (2005). Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959.

[Mendelzon and Wood, 1995] Mendelzon, A. and Wood, P. (1995). Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258.

[Miklau and Suciu, 2004] Miklau, G. and Suciu, D. (2004). Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45.

[Milo et al., 2002] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., and Alon, U. (2002). Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827.

[Neo4j, 2013] Neo4j (2013). Neo4j, The graph database. `http://www.neo4j.org/`.

[Neven, 2002] Neven, F. (2002). Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46.

[Neven et al., 2004] Neven, F., Schwentick, T., and Vianu, V. (2004). Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435.

[Olken, 2003] Olken, F. (2003). Graph data management for molecular biology. 7(1):75–78.

[Papadimitriou, 1993] Papadimitriou, C. H. (1993). *Computational Complexity*. Addison Wesley.

[Pérez et al., 2009] Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of sparql. *ACM Transactions on Database Systems*, 34(3).

[Pérez et al., 2010] Pérez, J., Arenas, M., and Gutierrez, C. (2010). nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270.

[Reutter, 2013a] Reutter, J. L. (2013a). Containment of nested regular expressions. CoRR abs/1304.2637.

[Reutter, 2013b] Reutter, J. L. (2013b). *Graph Patterns: Structure, Query Answering and Applications in Schema Mappings and Formal Language Theory*. PhD thesis, School of INformatics, University of Edinburgh.

[Ronen and Shmueli, 2009] Ronen, R. and Shmueli, O. (2009). Soql: a language for querying and creating data in social networks. In *25th International Conference on Data Engineering (ICDE)*, pages 1595–1602.

[Rudolph and Krötzsch, 2013] Rudolph, S. and Krötzsch, M. (2013). Flag & check: data access with monadically defined queries. In *PODS*, pages 151–162.

[Sakamoto and Ikeda, 2000] Sakamoto, H. and Ikeda, D. (2000). Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.*, 231(2):297–308.

[San Martín and Gutierrez, 2009] San Martín, M. and Gutierrez, C. (2009). Representing, querying and transforming social networks with rdf/sparql. In *6th European Semantic Web Conference (ESWC)*, pages 293–307.

[Schwentick, 2004] Schwentick, T. (2004). Xpath query containment. *SIGMOD Record*, 33(1):101–109.

[Segoufin, 2006] Segoufin, L. (2006). Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57.

[Segoufin, 2007] Segoufin, L. (2007). Static analysis of XML processing with data values. *SIGMOD Record*, 36(1):31–38.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing.

[Tal, 1999] Tal, A. (1999). Decidability of inclusion for unification based automata. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology.

[Tarski and Givant, 1987] Tarski, A. and Givant, S. (1987). *A Formalization of Set Theory Without Variables*. AMS.

[ten Cate, 2006] ten Cate, B. (2006). The expressivity of XPath with transitive closure. In *25th ACM Symposium on Principles of Database Systems (PODS)*, pages 328–337.

[ten Cate and Lutz, 2009] ten Cate, B. and Lutz, C. (2009). The complexity of query containment in expressive fragments of XPath 2.0. *Journal of the ACM*, 56(6).

[ten Cate and Marx, 2007] ten Cate, B. and Marx, M. (2007). Navigational XPath: calculus and algebra. *Sigmod Record*, 36(2):19–26.

[Van den Bussche and Vossen, 1993] Van den Bussche, J. and Vossen, G. (1993). An extension of path expressions to simplify navigation in object-oriented queries. In *DOOD*, pages 267–282.

[Vardi, 1982] Vardi, M. Y. (1982). The complexity of relational query languages. In *STOC*, pages 137–146.

[Vardi, 1995] Vardi, M. Y. (1995). On the complexity of bounded-variable queries. In *PODS*, pages 266–276.

[W3C Consortium, 2013] W3C Consortium (2013). Semantic web: The w3c consortium's vision of the web of linked data. `http://www.w3.org/standards/semanticweb/`.

[Wood, 2012] Wood, P. (2012). Query languages for graph databases. *Sigmod Record*, 41(1):50–60.

[Xpath, 1999] Xpath (1999). XML Path Language (XPath). www.w3.org/TR/xpath.

[Xpath 2.0, 2010] Xpath 2.0 (2010). XML Path Language (XPath) 2.0 (Second Edition). www.w3.org/TR/xpath20.

[Yang et al., 2008] Yang, L., Dang, Z., and Ibarra, O. H. (2008). On stateless automata and p systems. *International Journal of Foundations of Computer Science*, 19(5):1259–1276.

Index

FO*, 150
TriAL, 175
TriAL$^=$, 187
TrCl, 194, 201
c, 132
GXPath$_{\text{cond}}$, 151
#GXPath$_{\text{core}}$, 133
GXPath$_{\text{core}}$, 132
#GXPath, 133
GXPath, 132
GXPath$_{\text{core}}^{\text{path-pos}}$, 136
GXPath$_{\text{reg}}^{\text{path-pos}}$, 136
reachTA$^=$, 191
GXPath$_{\text{reg}}$, 132
$\sim$, 132
TripleDatalog$^\neg$, 178
eq, 132
2RPQ, 15
2RQB, 76
2RQD, 76
2RQM, 73

C2RPQ, 16
Conditional GXPath, 151
Conditions, 35
Conjunctive GXPath, 162
Conjunctive queries, 78
  CRDPQ, 78
  CRQB, 78
  CRQD, 78
  CRQM, 78
  CRQV, 78
Conjunctive regular path queries, 15
Core GXPath, 132
CRPQ, 15

Data graph, 10
Data path, 14
Data words, 86

Graph XPath, 132
Graph database, *see also* Data graph
Graph languages, 20
Ground RDF document, 168

Join, 174

Language containment, 87
Left Kleene closure, 176

Membership, 87

Navigational languages, 10
Nested path query, 17
Nested regular expressions, 17
Node expressions, 132
  Node formulas, 132
  Node tests, 132
Nonemptiness, 87
NPQ, 17
NRE, 17

Parameter-free Transitive-closure logic, 150
Path, 14
Path expressions, 132
  Path formulas, 132
Path languages, 20
Path-positive GXPath, 136
Positive GXPath, 136

Query answering, *see also* Query evaluation
Query containment, 227
Query evaluation, 18

RDF triple, 168
RDPQ, 37
Register automata, 35
  over data words, 89
Register automata with variables, 80
Regular GXPath, 132
Regular data path query, 37
Regular expressions with binding, 50
  over data words, 103
Regular expressions with equality, 56
  over data words, 115
Regular expressions with memory, 40
  over data words, 94

Regular path queries, 14
Regular queries with binding, 52
Regular queries with data tests, 59
Regular queries with memory, 45
Regular queries with variables, 66
Relation algebra, 147
REM, 94
REWB, 103
REWE, 115
right Kleene closure, 176
RPQ, 14
RQB, 52
RQD, 59
RQM, 45
RQV, 66

semipath, 16

Transitive closure logic, 194, 201
Triple Algebra, 175
Triple join, 174
Triplestore, 172
Two-way regular path queries, 15
Two-way regular queries with binding, 76
Two-way regular queries with data tests, 76
Two-way regular queries with memory, 73

Universality, 87
URI, 168

Variable automata, 64
    over data words, 123
varRA, 80
VFA, 64